VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Rozšiřující knihovna pro modeler neuronových sítí

Extension Library for Neural Net Modeler

2019 Bc. Vojtěch Pustówka

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Vojtěch Pustówka**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Rozšiřující knihovna pro modeler neuronových sítí
Extension Library for Neural Net Modeler

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je vytvořit prototyp knihovny pro paralelizaci vícevrstvých neuronových sítí s učením Back-propagation a Konvoluční neuronové sítě v jazyce Java. Tato knihovna je určena jako rozšíření existující aplikace Modeleru neuronových sítí.

Práce bude obsahovat:
1. Přehled aktuálního stavu paralelizace zpracování neuronových sítí.
2. Návrh knihovny.
3. Implementaci knihovny.
4. Experimenty pro porovnání různých přístupů.

Seznam doporučené odborné literatury:

[1] ROJAS, Raul. Neural networks: a systematic introduction. New York: Springer-Verlag, c1996. ISBN 3540605053.
[2] GOODFELLOW, Ian, Yoshua BENGIO a Aaron COURVILLE. Deep learning. Cambridge, Massachusetts: The MIT Press, [2016]. ISBN 9780262035613.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. David Ježek, Ph.D.**

Datum zadání: 01.09.2018

Datum odevzdání: 30.04.2019

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 23. dubna 2019 .....

Abstrakt

Tato diplomová práce se zabývá možnostmi distribuce výpočtů nad vícevrstvými neuronovými sítěmi a vytvořením prototypu knihovny, která problém distribuce výpočtů řeší. Dle získaných poznatků pak práce popisuje návrh a implementaci prototypu knihovny a popisuje také testování prototypu knihovny v různých prostředích včetně superpočítače. Prototyp knihovny je určen pro aplikaci Modeler neuronových sítí.

Klíčová slova: Vícevrstvá neuronová síť, Distribuovaný výpočet, Java, Paralelismus, Backpropagation

Abstract

This thesis focuses on possibilities of distributed computing of multilayer neural network and creating a prototype library which solves the distributed computing problem. According to gained knowledge, it describes the design and implementation of the prototype library and also describes testing of the prototype library in different environments including a supercomputer. The prototype library is intended to be used by application Neural Net Modeler.

Key Words: Multilayer neural network, Distributed computing, Java, Parallelism, Backpropagation

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Paralelní distribuce učení neuronových sítí se v čase ukázala jako nezbytná součást jejich používání. Neuronové sítě, stejně jako i jiné algoritmy vyžadující velký výpočetní výkon, nemusí být v jádru složité, leč jejich efektivní využití není uskutečnitelné bez možnosti rozprostření jejich běhu efektivně mezi výpočetní uzly. Doba kdy na celý výpočet stačil jeden procesor s jedním jádrem (i když taktovaným do závratných výšin) je dávno pryč. Moderní procesory už nemohou zvyšovat svůj výkon pouze taktováním jádra, proto je aktuální cestou zvyšování výkonu přidávání jader.

Přidáním jednoho jádra se samozřejmě výkon nezvedne dvakrát, ale umožní procesorům zvýšit svůj výpočetní výkon dost na to, abychom se na cestě za výkonem mohli posouvat dál. K podobnému problému a podobnému řešení došel také vývoj grafických jednotek. Tomuto nově dostupnému výkonu se ovšem musí uzpůsobit aplikace i algoritmy. Možným způsobem přizpůsobení je paralelní distribuce.

Jako jeden z nejlepších případů užití paralelní distribuce výpočtu se jeví učení neuronové sítě. Dosažení kvalitního naučení neuronové sítě trvá velmi dlouho a paralelizace výpočtu mezi jádry tento proces citelně urychlí.

Paralelní distribuce výpočtu na jádrech jednoho procesoru je ovšem pouze jeden ze způsobů, jak lze paralelizaci využít.

Dalším ze způsobů paralelizace je rozdělení výpočtů mezi samotné počítače, tedy zahrnutí více samostatných strojů do procesu výpočtu, v tomto případě učení. Tím, že do procesu učení zařadíme jak paralelizaci nad jádry procesoru, tak i paralelizaci výpočtu mezi počítače, můžeme docílit výrazného, ne-li zásadního, urychlení celého procesu. Stejně jako paralelní distribuce na jádrech procesoru i tato paralelizace sebou nese své výhody a nevýhody.

S nárůstem možností využití neuronových sítí v komerční sféře je kladen větší důraz na jejich efektivnost, a tudíž i na rychlost, se kterou jsou schopny se učit. Tím je samozřejmě kladen větší důraz i na možnosti paralelizace. Ideální řešení by nabízelo rychlou paralelizaci jak nad procesory, tak nad počítači. Dokázalo by zvládnout sítě všech velikostí a bylo schopné se optimálně přizpůsobit struktuře výpočetní sítě. Takové řešení ovšem není jednoduché vytvořit. Každá společnost má svá vlastní pravidla, podmínky a výpočetní sítě, na kterých své neuronové sítě provozuje. Uspokojit veškeré požadavky se tedy zdá jako nereálný úkol.

Projekt Modeler neuronových sítí slouží primárně jako ukázkové řešení fungování neuronových sítí za účelem jejich vysvětlení, a dále také pro testování a zkoušení prototypů aplikací využívajících neuronové sítě. V době psaní této práce podporoval Modeler neuronových sítí převážně sítě založené na metodě Backpropagation. Neuronové sítě založené na metodě Backpropagation se učí velmi pomalu a paralelizace je u nich nezbytná k dosažení reálné použitelnosti. Právě z tohoto důvodu vznikla motivace k napsání této práce a vytvoření knihovny pro představený projekt. Nová knihovna poslouží jako základ pro řešení paralelizace výpočtu neuronových sítí.

Informace o struktuře textu

Text této práce je strukturován do kapitol a podkapitol, které jsou uspořádány a číslovány standardním způsobem. Hlavní kapitoly začínají vždy na nové straně. Zdroje jsou v práci uváděny v hranatých závorkách obsahujících číslo, které se odkazuje do seznamu použité literatury. Není-li u vložených tabulek, grafů či obrázků uvedeno jinak, jedná se o dílo autora práce.

2 Cíle

Cílem této práce je vytvoření prototypu knihovny řešící paralelizaci vícevrstvých neuronových sítí. Knihovna bude připravená řešit sítě s učením typu Backpropagation a konvoluční neuronové sítě. Od knihovny se očekává, že bude efektivně řešit paralelizaci vícevrstvých neuronových sítí na úrovni komunikace mezi samostatnými počítači, bude obsahovat jednoduché a přehledné rozhraní, přes které se napojí cílová neuronová síť a bude disponovat možností knihovnu jednoduše nastavit pomocí základních konfiguračních prvků.

Součástí této práce bude také analýza aktuálního stavu knihoven a nástrojů pro paralelizaci neuronových sítí, které jsou v průběhu tvoření této práce dostupné. Práce bude zkoumat způsoby paralelizace, které dostupné knihovny používají, porovnávat jejich přístup, a pak z nich vyvodí směr, kterým se bude prototyp knihovny ubírat.

3 Analýza aktuálního stavu

Zpracování vícevrstvých neuronových sítí pomocí paralelizace není novou myšlenkou. Zefektivnění jejich práce je cílem autorů už od počátku jejich používání. Pochopitelně existuje mnoho různých přístupů, jak vícevrstvé neuronové sítě paralelizovat a zefektivnit. Tato kapitola se zabývá rozborem možností paralelizace a metodami, které jsou dostupné pro komerční či akademické využití.

3.1 Dostupné metody paralelizace

Tato práce se spíše než paralelizací výpočtu nad jádry CPU či GPU zabývá paralelizací v síti dvou anebo více samostatných výpočetních jednotek. CPU i GPU jednotky učinily v posledních letech značný pokrok v oblasti výkonu, ale data, která musí vícevrstvé neuronové sítě dnes zpracovávat, jsou tak obrovská, že učení těchto neuronových sítí by zabralo až nepřijatelné množství času. To znemožňuje praktické využití vícevrstvých neuronových sítí.

Není vůbec těžké si představit, že porozumění jazyku, rozeznání obrazů nebo třeba rozpoznání podvodných aktivit vyžaduje tak obrovské množství dat, že učení sítě pouze na jednom zařízení není reálné.

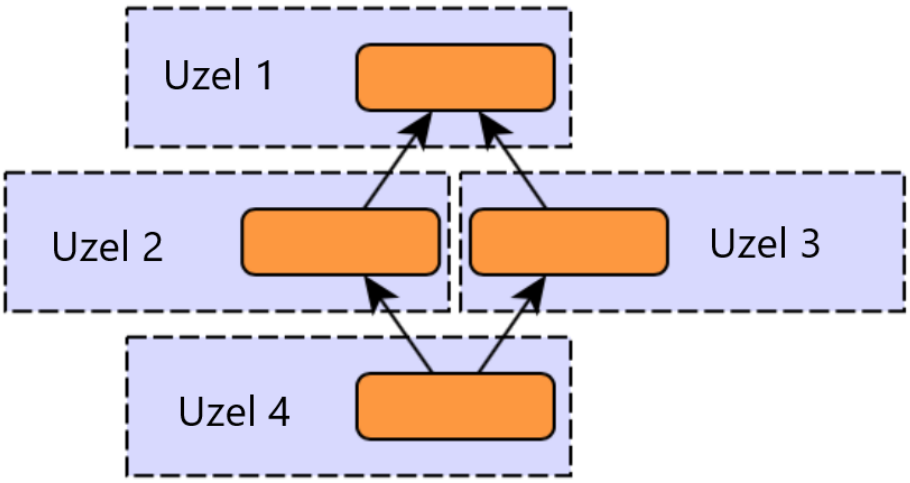
Byla provedena spousta pokusů a výzkumů zabývajících se možností distribuce výpočtů nad vícevrstvými neuronovými sítěmi. Nejčastěji se o distribuovaných výpočtech mluví ve spojitosti s dvěma modely paralelismu. A to modelovým paralelismem a datovým paralelismem.

3.2 Modelový paralelismus

Modelový paralelismus představuje spolu s datovým paralelismem dva nejpopulárnější alternativní přístupy k řešení paralelních výpočtů nad vícevrstvými neuronovými sítěmi. V kontextu neuronových sítí si lze modelový paralelismus představit jako jednu velkou síť, která je rozprostřená po všech strojích účastnících se distribuovaného výpočtu. Tedy celá soustava výpočetních uzlů představuje jednu síť, a jednotlivé výpočetní uzly se starají o jednotlivé části sítě. Mohlo by se jednat například o jednotlivé vrstvy, nebo možná i jednotlivé skupiny neuronů.

Modelový paralelismus ve své ideji zavádí myšlenku, že je možné nezávisle paralelizovat vrstvy či skupiny neuronů. Často tato metoda ovšem naráží na zásadní problém, že modelový paralelismus nepočítá se závislostí jednotlivých vrstev na ostatních vrstvách (či skupin neuronů na jiných skupinách) a místo aby se výpočet prováděl zcela nezávisle (a tedy paralelně) na všech vrstvách, čekají zapojené uzly na výsledky z ostatních vrstev. Způsobuje to fakt, že standardní architektura vícevrstvých neuronových sítí vytváří závislosti mezi vrstvami, a tím generuje zbytečnou komunikaci. Tak dochází ke zpomalení celého výpočtu a porušení principu distribuovaného výpočtu.

Ukázku modelového paralelismu lze vidět na obrázku 1.



Obrázek 1: Modelový paralelismus

3.2.1 Řešení problému modelového paralelismu

Aby bylo možné efektivně používat modelový paralelismus, je třeba si poradit s problémem generování zbytečné komunikace mezi vrstvami.

Jedním ze způsobů, jak toto lze řešit, je zavedení redundantních výpočtů do neuronové sítě. To lze ideálně provést dělením sítě, kdy se každý procesor stará o dvojnásobek množství neuronů a tudíž sice musí počítat více, ale komunikuje méně.

Druhým možným způsobem je využití Cannonova algoritmu pro multiplikaci matic. Není možné v rámci této práce dokázat výhody tohoto řešení při použití v modelovém paralelismu, ale existují důkazy, které naznačují, že užití Cannonova algoritmu upraveného pro použití ve vícevrstvých neuronových sítích pozitivně ovlivňuje efektivnost a rychlost dělení na malých, plně propojených neuronových sítích.[2]

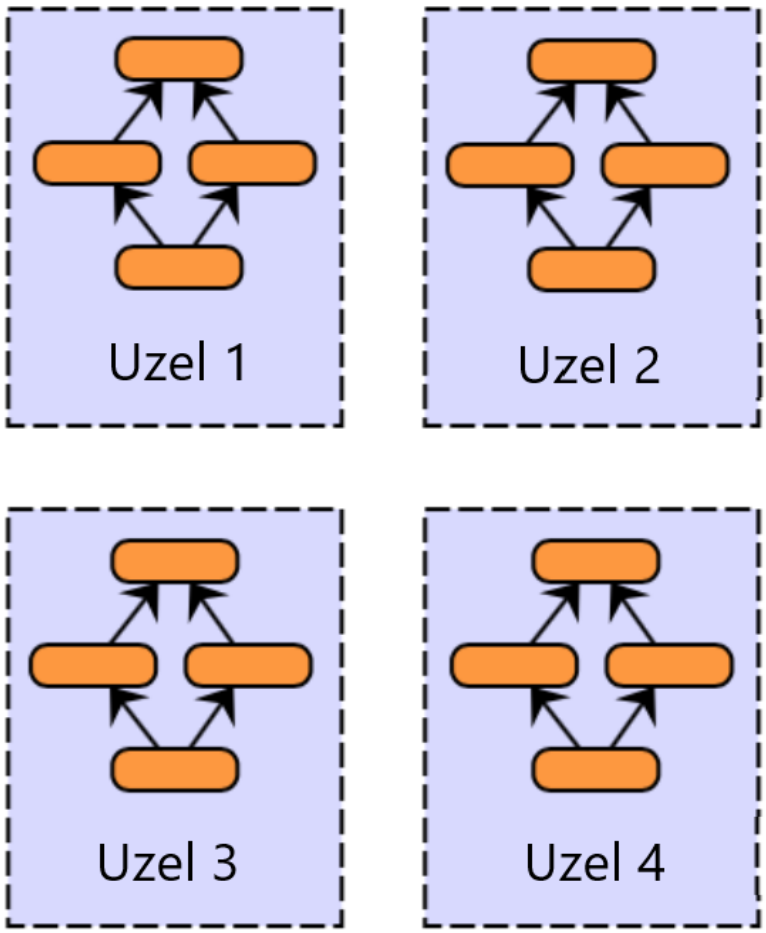
3.2.2 Zhodnocení modelového paralelismu

Modelový paralelismus přináší výhody obzvlášť v oblasti škálovatelnosti pro velké modely, ale zároveň sebou nese nepříjemnou nutnost vyřešit zbytečnou komunikaci vznikající mezi vrstvami neuronové sítě.

3.3 Datový paralelismus

Datový paralelismus představuje oproti modelovému paralelismu opačný přístup k paralelním výpočtům nad vícevrstvými neuronovými sítěmi. Zatímco modelový paralelismus pohlížel na neuronovou síť jako na jeden velký model a pokoušel se její části namapovat na dostupné výpočetní uzlu, datový paralelismus pohlíží na celou neuronovou síť jako na skupinu nezávislých modelů. V datovém paralelismu má každý samostatný výpočetní uzel svou vlastní kopii modelu (tedy neuronové sítě) a data pro výpočet se rovnoměrně rozdělují mezi uzly. Na konci výpočetního cyklu pak uzly spočtená data složí do konečného výsledku.

Příklad datového paralelismus lze vidět na obrázku 2.



Obrázek 2: Datový paralelismus

Datový paralelismus se ve své myšlence efektivně brání nutnosti řešit závislosti mezi vrstvami neuronové sítě tím, že každému uzlu předá jednu celou kopii sítě. Naopak však musí řešit efektivní distribuci dat potřebných pro výpočty neuronových sítí, zabývat se funkčním způsobem sesbírání spočtených dat z jednotlivých výpočetních uzlů a jejich složením do podoby finálního výsledku. Lze si tedy snadno uvědomit, že rychlost výpočtů pomocí datového paralelismu závisí více na rychlé komunikaci mezi či s uzly než na rychlosti výpočtu v uzlu samotném. Tím, že každý uzel obsahuje svou vlastní kopii sítě, není problém celý průběh výpočtu paralelizovat.

3.3.1 Řešení problémů datového paralelismu

Problém rychlé komunikace mezi výpočetními uzly, tedy rychlého předávání menšího množství dat potřebného pro výpočet neuronových sítí, je napojen na dvě klíčové vlastnosti komunikace jako takové mezi účastníky. Těmi jsou rychlost a spolehlivost. S nároky na vysokou rychlost pochopitelně klesá úměrně spolehlivost, a naopak. Proto rychlost komunikace mezi výpočetními uzly obsahujícími jednotlivé neuronové sítě lze snadno ovlivnit volbou správného komunikačního protokolu, a s touto volbou je pak nutné řešit i její následky.

Problém sbírání dat vyžaduje složitější řešení. Kvůli potřebě zefektivnit chod výpočtu neuronových sítí jako celku by bylo ideální, kdyby se jednotlivé sítě neučily samostatně, ale byly by schopné si mezi sebou předávat průběžné výsledky svých výpočtů, a tedy mít možnost zlepšit učení využitím dat získaných učením sítí v předcházejících iteracích. Jako řešení tohoto problému se nabízejí dva přístupy. Těmi jsou vážení parametrů a upravování hodnot na základě gradientu. Výběr jednoho z těchto přístupu by měl být založen na základě podrobnějšího návrhu.

3.4 Použití modelového i datového paralelismu současně

Není nutné zvolit pouze jednu z výše zkoumaných možností. Je možné datový i modelový paralelismus používat současně, protože jednotlivé metody se nevylučují. Šlo by ku příkladu mít na každém výpočetním uzlu jednu kopii vícevrstvé neuronové sítě, jak ukazuje datový paralelismus, a zároveň síť rozdělit po vrstvách či skupinách neuronů na jednotlivá výpočetní jádra dostupných CPU či GPU.

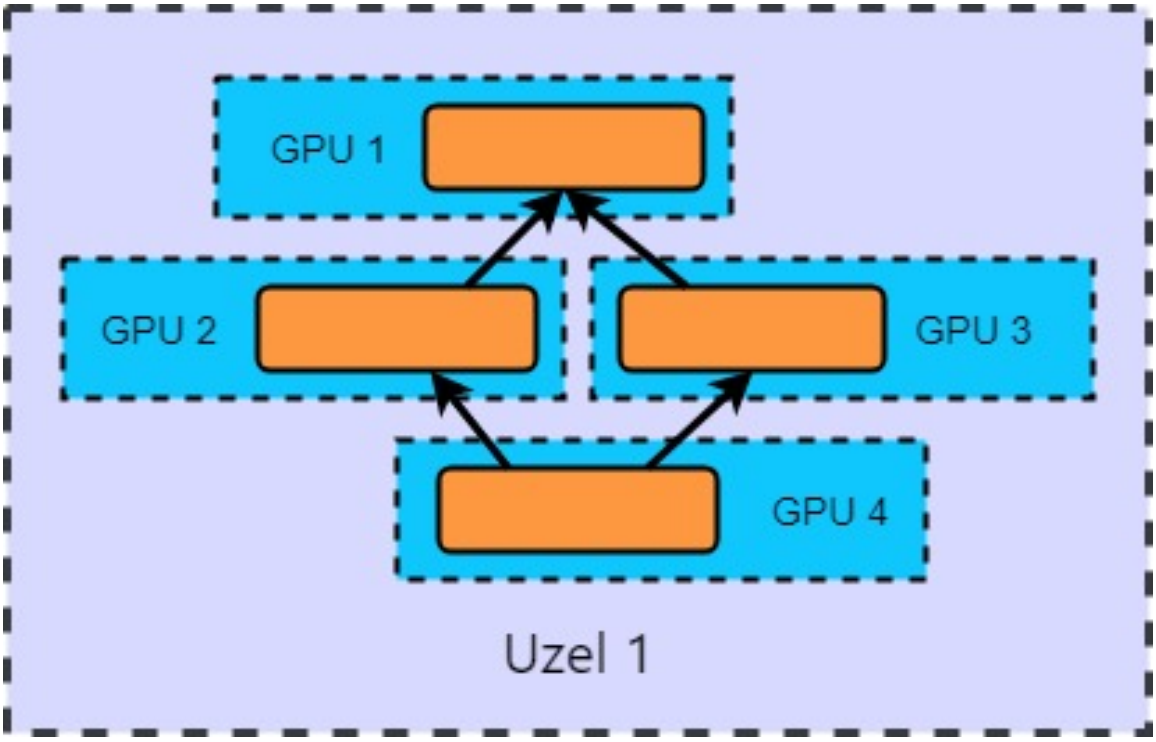
Ukázkový příklad kooperace datového a modelového paralelismu lze vidět na obrázku 3.

3.5 Volba metody paralelizace

Z poznatků získaných v předchozích podkapitolách lze usoudit, že vhodnější metodou pro řešení distribuovaného výpočtu je metoda datového paralelismu.

Řešení problémů spojených s rychlostí komunikace a přeposíláním dat mezi uzly se ukázalo jako jednodušší oproti eliminaci problému s nadbytečnou komunikací mezi vrstvami či skupinami neuronů.

Tím, že byl ve fázi výběru metody paralelismu vybrán datový paralelismus, bude nutné vyřešit problémy, které spolu datový paralelismus přináší, a které byly nastíněné v podkapitole 3.3.



Obrázek 3: Datový a modelový paralelismus v kooperaci

Ve zkratce se jedná o řešení efektivní komunikace a předávání mezivýsledků mezi jednotlivými neuronovými sítěmi.

3.6 Způsob distribuce mezivýsledků mezi výpočetními uzly

Jak bylo naznačeno výše, existují dva hlavní přístupy, které řeší efektivní distribuci mezivýsledků mezi výpočetními uzly (tedy kopiemi vícevrstvých neuronových sítí). Mluvíme o metodě vážení parametrů a upravování hodnot na základě gradientu.

Prototyp knihovny bude využívat datový paralelismus, a proto musí distribuovat data k jednotlivým uzlům. Každý výpočetní uzel následně provede samostatný výpočet, kterým je zde myšleno hlavně učení, a získá tak mezivýsledek.

Pro ještě efektivnější výpočet se jeví jako ideální, aby jednotlivé výpočetní uzly distribuovaly své mezivýsledky všem dalším uzlům. Takto není nutné, aby pokaždé začínal proces výpočtu takzvaně od nuly, ale mohl se postupně zlepšovat na základě dat poskytnutých uzly během výpočtů v předchozích iteracích. Každý uzel, přesněji každá kopie neuronové sítě, by měla obsahovat mechanismy pro distribuci získaných mezivýsledků a akceptaci mezivýsledků ostatních uzlů, které použije jako základ pro další výpočet.

3.6.1 Distribuce mezivýsledků metodou vážení parametrů

Řešení distribuce mezivýsledků metodou vážení parametrů patří k nejjednodušším přístupům k distribuci dat v datovém paralelismu pro neuronové sítě. Lze si ji jednoduše představit a pochopit v pěti krocích.

1. Inicializace parametrů neuronových sítí pomocí náhodných hodnot.
2. Distribuce hodnot jednotlivým uzlům.
3. Výpočet (učení) jednotlivých uzlů.
4. Nastavení globálních parametrů na hodnotu získanou vážením mezivýsledků.
5. Dokud existují nezpracovaná data, pokračování krokem 2.

Z výše popsaných kroků je zřejmé, že popsaná podoba metody vážení parametrů je centralizovaná. Tedy existuje určitý globální prvek, který se stará o distribuci dat a sbírá mezivýsledky z jednotlivých výpočetních uzlů. Přestože neprovádí žádný výpočet, udržuje si také kopii neuronové sítě, která tak slouží jako globální držitel dat. Představuje výsledek celého distribuovaného výpočetního procesu, upravuje se všemi získanými daty a její vlastní data se používají jako inicializační hodnoty pro výpočet jednotlivých uzlů. Tento globální prvek si lze z pohledu komunikace představit jako Master prvek a jednotlivé výpočetní uzly jako Slave prvky.

Vážení parametrů lze zapsat jednoduchou rovnicí 1.

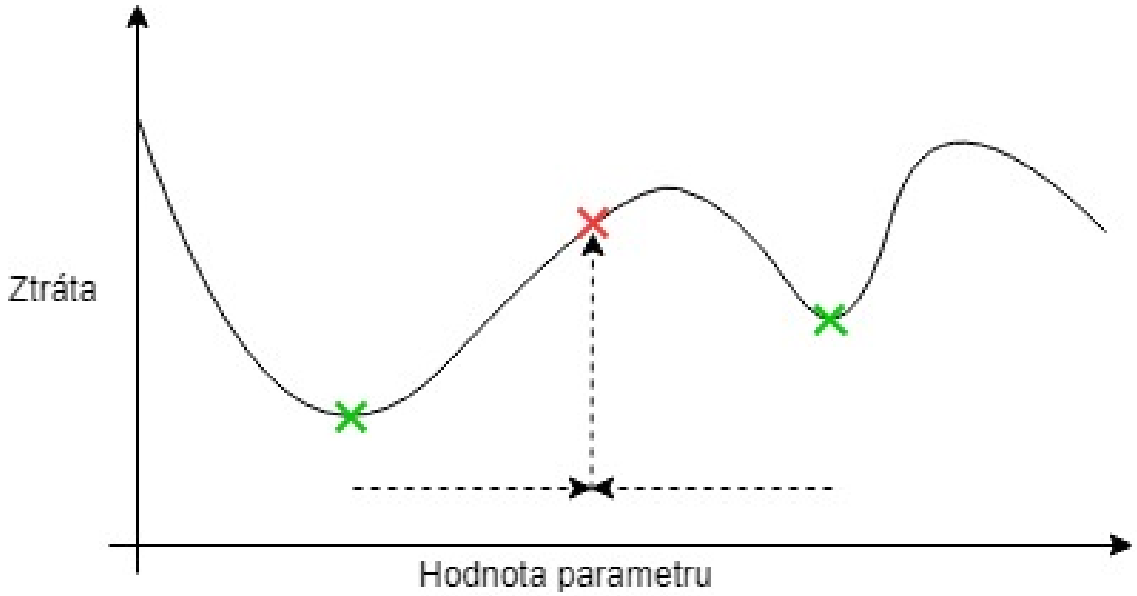
$$W_{i+1} = \frac{\alpha}{nm} \sum_{j=1}^{mn} W_{i+1,j} \tag{1}$$

W_i představuje hodnotu váhy na pozici i, n představuje množství neuronů, m je počet vzorů (dat k učení) a α je poměr učení. Pak $m * n$ se rovná úplnému počtu zpracovaných vzorů mezi jednotlivými váženími parametrů.

Tato rovnice jasně ukazuje, že jednotlivé výpočetní uzly předávají řídícímu globálnímu uzlu hodnoty vah získaných při výpočtu, a ten je následně využívá k upravení hodnot vah globální neuronové sítě.

Takto popsaný postup naznačuje, že je možné vždy provést jednu iteraci, a po jejím konci rozeslat data a upravit parametry vážením. Pokud se bude postupovat takto, lze se dopracovat ke správnému výsledku, ovšem cena za neustálou komunikaci a přepočítávání může narůst do takových rozměrů, že převáží výhody plynoucí z použití distribuované architektury.

Vážení parametrů je nejčastěji implementováno s vážící periodou N, která představuje množství dat (počet dávek) tak, že $N > 1$. Pokud ovšem bude N nastaveno na hodnotu příliš vysokou, a budeme tedy vážit příliš málo často, můžou se parametry neuronových sítí jednotlivých výpočetních uzlů lišit natolik, že ve výsledku bude centrální model těžko použitelný. Je nutné si uvědomit, že vážením N různých lokálních minim nemusí být lokální minimum. To lze vidět na obrázku 4



Obrázek 4: Ztráta při vážení parametrů

Správná nebo špatná volba vážící periody může fatálně ovlivnit to, jak efektivně bude distribuovaný výpočet dané neuronové sítě probíhat. Neexistuje úplně jasná odpověď na to, jak správně vážící parametr zvolit a zodpovězení této otázky je nadále komplikováno faktem, že se do vážení promítají parametry ovlivňující učení neuronové sítě, jako jsou poměr učení, velikost dávky dat nebo počet zapojených výpočetních uzlů.

3.6.2 Centralizovaný asynchronní stochastický sestup gradientu

Alternativním přístupem k výměně dat mezi uzly v datovém paralelismu je centralizovaný asynchronní stochastický sestup gradientu (Async SGD), dále jenom centralizovaný sestup gradientu. Centralizovaný sestup gradientu je v jádru podobný metodě vážení parametrů. Stejně jako metoda vážení vyžaduje globální prvek, který si udržuje kopii vícevrstvé neuronové sítě a uchovává si tak globální stav.

Na rozdíl od metody vážení parametrů nepředává síť mezivýsledky v podobě spočtených hodnot vah, ale jako gradienty hodnot. Tedy rozdíly mezi hodnotami vah před a po spočítání.

Centralizovaný asynchronní stochastický sestup gradientu lze popsat pomocí následující rovnice 2.

$$W_{i+1} = W_i - \lambda \sum_{j=1}^{n} \Delta W_{i,j} \tag{2}$$

W představuje hodnotu váhy na pozici i, λ představuje faktor škálování a n značí počet výpočetních uzlů.

Velkou výhodu oproti metodě vážení parametrů získává centralizovaný sestup gradientu právě tehdy, když se upustí od synchronizace a umožní se asynchronizace. Asynchronizace dovoluje upravit hodnoty vah neuronové sítě centrální jednotky hned poté, co některý z výpočetních uzlů dodá mezivýsledky. Není nutné čekat na to, až jednu celou iteraci dokončí všechny výpočetní uzly a až následně aplikovat získané mezivýsledky. Tato asynchronizace s sebou nese dva pozitivní vlivy:

- zvýšení propustnosti v distribuovaném systému (a tak mohou jednotlivé uzly trávit více času prováděním výpočtu než čekáním na data),
- jednotlivé uzly mohou přijít k aktuální datům od ostatních uzlů dřív než až na začátku další iterace.

3.6.3 Problém zastaralého gradientu (stale gradient problem)

Výše zmíněné pozitivní vlivy mají bohužel svou cenu. Zavedením asynchronizace zavádíme také problém zastaralého gradientu. Problém zastaralého gradientu je ve své podstatě velmi jednoduchý. Pokaždé když od jednoho výpočetního uzlu přijdou nové gradienty, musí se provést úprava hodnot globální neuronové sítě. Než ovšem stihl výpočetní uzel provést svůj výpočet, mohly být centrální hodnoty upraveny již mnohokrát.

Pokud je tento problém neřešen, může docházet k velkému zastarání gradientu. Je dokázáno, že máme-li N výpočetních uzlů, které provádějí své výpočty, průměrné zastarání gradientu je rovno N.[1]

Většina reálných aplikací centrálního sestupu gradientu zachovává podobný přístup, jako byl popsán výše, ale liší se právě v přístupu k problému zastaralého gradientu. Mezi řešení problému zastaralého gradientu patří:

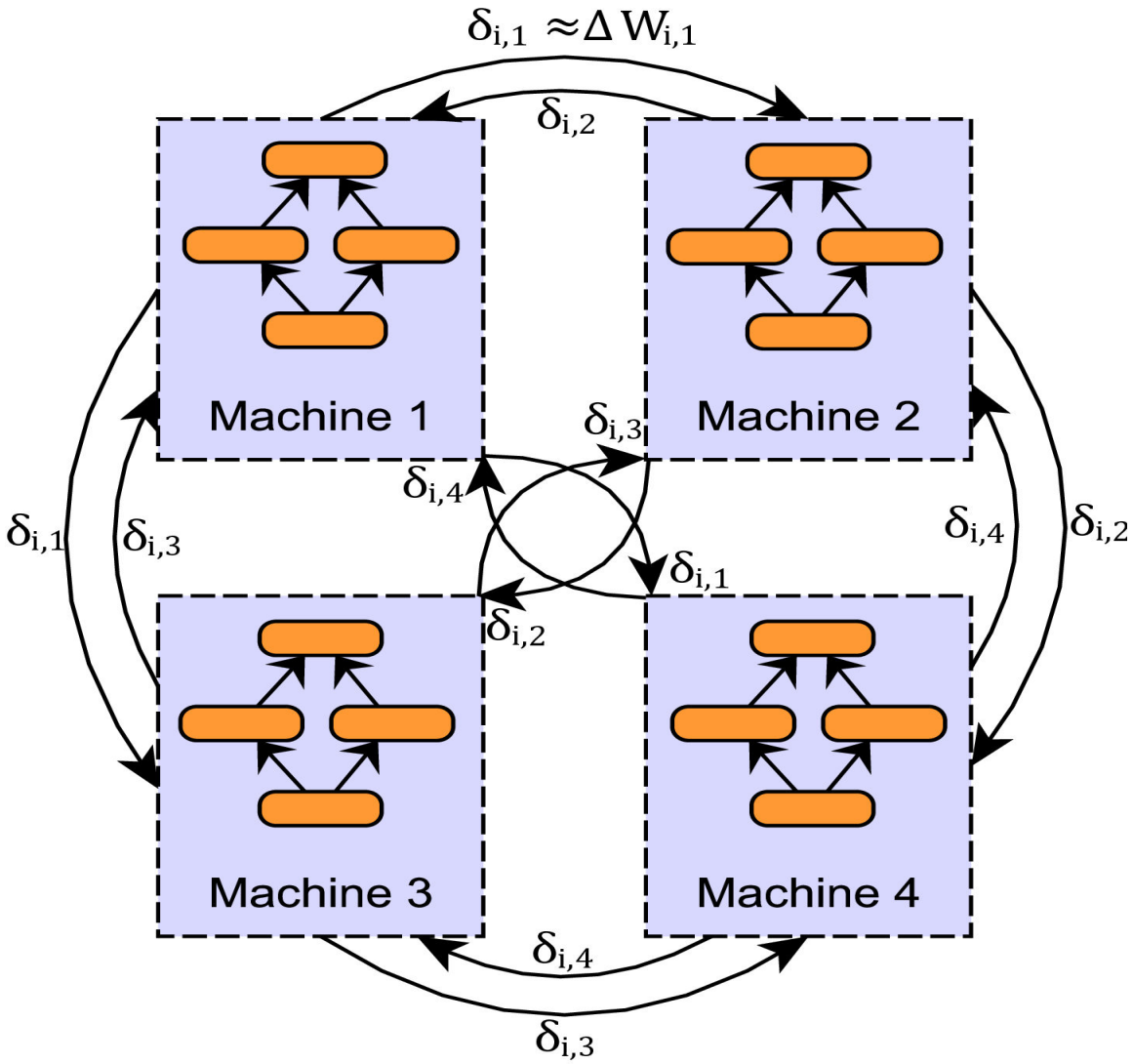
- škálování parametru λ nezávisle pro každé $\Delta W_{i,j}$ v závislosti na zastaralosti gradientu,
- implementace některého z 'měkkých' synchronních algoritmů,
- použití synchronizace k omezení zastarávání (například systém, kdy se rychlejší výpočetní uzly záměrně zpomalují, aby zastaralost gradientu nepřesáhla definovanou mez).

3.6.4 Decentralizovaný asynchronní stochastický sestup gradientu

Decentralizovaný asynchronní stochastický sestup gradientu představuje upravený přístup výše zmíněného centralizovaného asynchronního stochastického sestupu gradientu k datovému paralelismu. Na rozdíl od centralizovaného přístupu se musí vypořádat s absencí centrálního řídícího prvku. Další rozdíl představuje fakt, že model decentralizovaného asynchronního stochastického sestupu gradientu se musí obejít bez rozesílání dat a globální sítě udržující aktuální data. Tudíž si každý samostatný výpočetní uzel drží data potřebná pro svůj výpočet. Distribuce mezivýsledků stále probíhá, ale místo toho, aby je sbíral centrální prvek, tak si je mezi sebou předávají

všechny uzly navzájem. Rozdíl oproti centralizovanému přístupu se tedy projeví tak, že každý uzel si spravuje své data a provádí úpravu hodnot pomocí mezivýsledků, které obdrží od všech ostatních uzlů.

Oproti centralizovanému asynchronnímu stochastickému sestupu gradientu tak decentralizovaná podoba není závislá na jednom centrálním prvku, jehož selhání vede k selhání výpočtu jako takového, ale převádí funkčnost na jednotlivé výpočetní uzly. Tím se snižuje pravděpodobnost neúspěšného konce z důvodu pádu jednoho z uzlů, ale zároveň se zvyšuje množství komunikace mezi uzly z důvodu nutné výměny mezivýsledků. Zároveň si každý uzel musí uchovávat svou vlastní aktuální síť a provádět její aktualizaci pomocí vážení parametrů z ostatních sítí.



Obrázek 5: Schéma decentralizovaného asynchronního stochastického sestupu gradientu [4]

Ve standardním řešení datového paralelismu (tedy za použití buď vážení parametrů nebo centralizovaného asynchronního stochastického sestupu gradientu) se velikost přenášených dat v komunikaci rovná velikosti vektoru parametrů. Tedy buď přenášíme vektor nových parametrů nebo vektor gradientů (pro každý parametr jeden gradient).

Velké výhody přenášení upravovaného vektoru $\delta_{i,j}$ jsou následující:

- pouze část gradientu se přenáší v každém vektoru $\delta_{i,j}$ (o ostatních předpokládáme, že jsou nulové),
- lze je zakódovat pomocí integer indexu.

3.7 Dostupná řešení distribuovaného výpočtu nad vícevrstvými neuronovými sítěmi

K posouzení nejlepších přístupů k paralelizaci je vhodné prozkoumat i řešení, která používají již běžící a fungující knihovny či jiný dostupný software. Knihoven nebo jinak dostupného softwaru, které by poskytovaly komplexní a použitelné řešení distribuovaných výpočtů nad vícevrstvými neuronovými sítěmi, není mnoho. Existuje pouze malé množství takovýchto knihoven, dostupných v době psaní této práce, a pouze dvě z nich byly volně ke stažení a použití. Jedná se o knihovny TensorFlow a Deeplearning4j.

Knihovna TensorFlow v době psaní této práce neobsahovala kompletní řešení distribuovaných výpočtů na vícero uzlech (teprve se na tom pracovalo), proto není pro tuto práci přínosná. Oproti tomu Deeplearning4j obsahuje řešení distribuce výpočtu jak přes GPU či CPU, tak i na vícero samostatných výpočetních uzlech. Deeplearning4j je dostupná pod licencí Apache 2.0.

3.7.1 Deeplearning4j[5]

Deeplearning4j je volně dostupná knihovna (s otevřeným kódem) srovnatelná s komerčními řešeními sloužícími k distribuovanému učení vícevrstvých neuronových sítí od společnosti Skymind. Deeplearning4j využívá ve své implementaci distribuovaného učení datový model paralelismu využívající synchronní metodu vážení parametrů. Celý tento proces je implementován za pomocí technologie Apache Spark. Knihovna využívá jako centrální prvek Spark master node a jako výpočetní uzly Spark worker. Celý proces využívá pouze jeden centrální prvek. Pro lepší představu je proces popsán v následujících krocích.

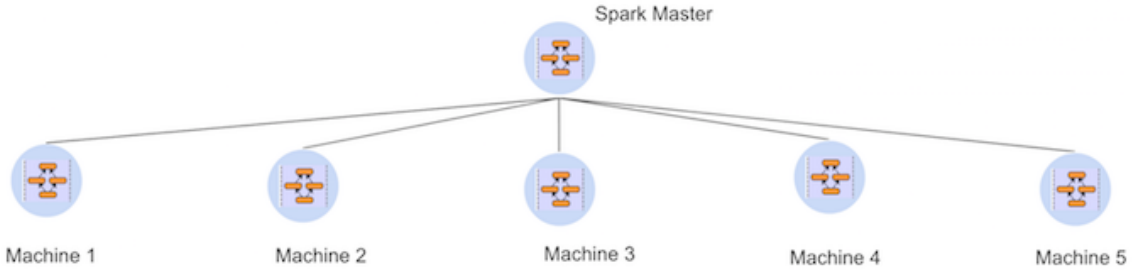
1. Master uzel (Spark ovladač) započne s iniciální konfigurací a parametry.
2. Data jsou rozdělena do množství menších celků na základě konfigurace v master uzlu.
3. Proces iteruje přes části dat. Rozešle konfiguraci, parametry z master uzlu do worker uzlů. Každému worker uzlu předá část dat. Provede vážení parametry a vrátí vážené výsledky do master uzlu.

4. Učení je dokončeno, master uzel má kompletní kopii naučené neuronové sítě.

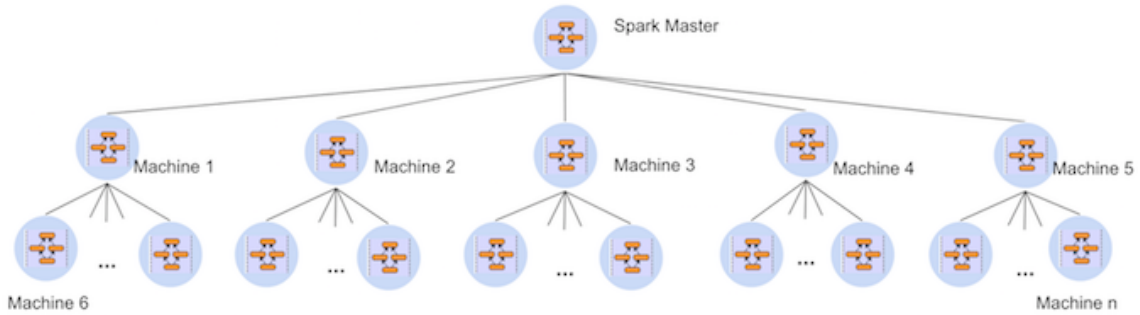
Samotní autoři uvádějí, že tato implementace, ač je kompletní a funkční, obsahuje značné množství míst, která je možné vylepšit, a tak celý proces učení urychlit.

Kromě vážení parametrů obsahuje knihovna i řešení distribuovaných výpočtů využívajících datový model paralelismu s metodou centralizovaného asynchronního stochastického sestupu gradientu. Toto řešení ovšem v době psaní této práce bylo v testovací beta verzi a výslovně nebylo určeno do produkčních prostředí.

K implementaci této varianty byl také použit Apache Spark. Pro organizaci výpočteních uzlů sdílejících mezi sebou gradienty využívá Deeplearning4j architekturu nazvanou plain mode (viz obrázek 6) nebo mesh mode (viz obrázek 7). Zatímco plain mode architektura představuje přiměřeně stabilní řešení, mesh mode architektura je zatím pouze experimentální řešení.



Obrázek 6: Deeplearning4j plain mode architektura[5]



Obrázek 7: Deeplearning4j mesh mode architektura[5]

3.8 Volba nejlepšího přístupu k datovému paralelismu

V předchozích podkapitolách bylo představeno množství různých přístupů k datovému paralelismu. Nelze jednoznačně prohlásit, že jedna z uvedených metod je absolutně dominantní a ostatní s přehledem překoná. Všechny představené metody mají své výhody a nevýhody, proto není jednoduché zvolit správnou metodu. Při výběru je možné se řídit splněním kritérií, která jsou kladena na distribuovaný výpočet a neuronovou síť. Následují vybraná kritéria, která jsou po neuronové síti a jejím distribuovaném výpočtu požadována:

- rychlost výpočtu (učení), například nejvíce naučených sad dat za danou jednotku času či nejnižší čas strávený učením během jedné epochy,
- nejvyšší udržitelná přesnost při neomezení počtu epoch, tedy při učení vedoucímu ke snížení nepřesnosti pod předem danou mez,
- nejvyšší udržitelná přesnost při v předem daném výpočetním čase,
- nejvyšší udržitelná přesnost při daném N počtu epoch.

Dále bude zcela jistě záležet i na parametrech neuronové sítě, jejíž výpočet bude paralelně distribuován. Mezi ty patří:

- typ neuronové sítě,
- počet vrstev vícevrstvé neuronové sítě,
- typ výpočetních uzlů používaných při učení neuronové sítě,
- druh komprese použité při přenášení dat (bude-li použita nějaká),
- učící algoritmus a jeho parametry.

3.8.1 Shrnutí synchronního vážení parametrů

Po definování všech těchto požadavků je potřeba dojít k rozhodnutí, která z metod bude použita v prototypu knihovny. Synchronní vážení parametrů vítězí v přesnosti učení na jednu epochu a celkově v udržitelnosti přesnosti, obzvlášť pokud je vážící perioda malá. Ale samozřejmě přidaná cena, kterou představuje synchronizace metody, znamená, že synchronní vážení parametrů je pomalejší v čase za periodu.

Tím, že se použije na propojení jednotlivých výpočetních uzlů kvalitní rychlá kabeláž, můžeme považovat synchronní vážení parametrů za stále použitelnou variantu. Můžeme také zapojit určitou formu komprese, a tak přenosovou rychlost ještě zvýšit.

3.8.2 Shrnutí centralizovaného asynchronního stochastického sestupu gradientu

Asynchronní stochastický sestup gradientu je vhodnou volbou pro učení a z počtu praktických užití lze odvodit, že je funkční volbou i v praxi. To samozřejmě platí za předpokladu, že je úspěšně vyřešen problém zastaralého gradientu.

Implementace asynchronního stochastického sestupu gradientu využívajícího centrálního prvku, tedy centralizovaná podoba, s sebou může přinést problém komunikačního úzkého hrdla. Tento problém lze ale jednoduše vyřešit použitím N centrálních prvků tak, aby celková výpočetní struktura udržovala specifický poměr výpočetních uzlů na centrální prvky.

3.8.3 Shrnutí decentralizovaného asynchronního stochastického sestupu gradientu

Decentralizovaný asynchronní stochastický sestup gradientu zní jako velmi slibný přístup, ale v tuto chvíli k němu neexistuje mnoho informací a v praxi se s ním lze setkat pouze v experimentálních implementacích. Lze ovšem převzít spoustu nápadů, které přišly s decentralizovaným asynchronním stochastickým sestupem gradientu a využít je v centralizované podobě řešení.

3.8.4 Volba řešení datového paralelismu

Z poznatků získaných analýzou přístupů k metodě datového paralelismu vychází jako nejvhodnější varianta pro prototyp vznikající knihovny centralizovaný asynchronní stochastický sestup gradientu. Odstraňuje omezení synchronizace, která s sebou nese synchronní vážení parametrů, ale oproti decentralizovanému asynchronnímu stochastickému sestupu gradientu si ponechává možnost využívat centrální prvek s primární kopií vícevrstvé sítě obsahující konečný výsledek. Zároveň se jedná o prozkoumanější metodu jak z pohledu výzkumu, tak komerčního využití.

4 Návrh knihovny

V této kapitole je popsán postupný návrh prototypu knihovny pro distribuovaný výpočet vícevrstvých neuronových sítí. Kapitola je zaměřena na výběr technologií a knihoven potřebných pro implementaci knihovny. Dále také řeší návrh implementace neuronových sítí založených na metodě Backpropagation.

4.1 Programovací jazyk

Prototyp knihovny bude implementován v jazyce Java.

Jazyk Java je velmi rozšířený, platformě nezávislý programovací jazyk. Java byla vytvořena Jamesem Goslingem v devadesátých letech minulého století. James Gosling při návrhu Javy vycházel z jazyka C++, a to proto, aby byl přechod na Javu pro vývojáře používající C++ jednodušší. Java je stejně jako C++ objektově orientovaný jazyk a je navržená tak, aby měla co nejméně závislostí na použité platformě. Tento princip se nazývá WORA (Write once, run anywhere) - napiš jednou, spusť kdekoliv. Javovské aplikace se standardně překládají do bytecodu, který běží na JVM (Virtuální Java Stroj) nezávisle na použité počítačové platformě.

Jazyk Java byl vybrán kvůli snadnému spojení s aplikací Modeler neuronových sítí, která je také napsaná v Javě. Dále také proto, že Java může běžet nezávisle na platformě a knihovna tedy nebude omezena operačním systémem nebo třeba počítačovou architekturou.

Použita bude Java SE ve verzi 11.0.2 (LTS). Jedná se o aktuální verzi Javy s dlouhodobou podporou.

4.2 Síťová komunikace

Prototyp knihovny bude muset velmi efektivně řešit síťovou komunikaci. V analýze funkčnosti datového modelu s použitím řešení pomocí centralizovaného asynchronního stochastického sestupu gradientu bylo jasně zjištěno, že i perfektně navržená architektura distribuovaného výpočtu nad vícevrstvou neuronovou sítí může vést k horším výsledkům než nedistribuovaný výpočet, pokud se dostatečně správně nenavrhne komunikace mezi výpočetními uzly.

Při špatném návrhu je možné dostat se do situace, kdy komunikace mezi uzly je tak pomalá, že anuluje veškerý zisk plynoucí z distribuovanosti výpočtu.

Mluvíme-li o komunikaci po síti, mluvíme ve většině případů o dvojici protokolů. Těmi jsou TCP a UDP.

4.2.1 TCP (Transmission Control Protocol)

TCP je protokol definující způsob, jakým navázat a udržet síťovou komunikaci, kterou mohou aplikace využívat pro vzájemnou výměnu dat. TCP spolupracuje s internetovým protokolem IP, který definuje, jak počítače mezi sebou posílají pakety dat. TCP bylo definováno IETF (Internet Engineering Task Force).

TCP je protokol orientovaný na spojení (connection-oriented), což znamená, že spojení je navázáno a udržováno dokud oba účastníci komunikace neoznačí komunikaci za uzavřenou. TCP je také nazýváno "spolehlivým" protokolem, protože garantuje, že data dorazí do cíle kompletní a v pořádku.

TCP rozhoduje o tom, jak rozdělit odesílanou zprávu do paketů, které síť dokáže doručit, rozesílá, přijímá a odesílá pakety na síťovou vrstvu ISO/OSI modelu, řídí komunikaci a obstarává její bezchybovost pomocí znovurozesílání ztracených nebo poškozených paketů. Stejně tak potvrzuje správné přijetí paketů druhé straně.

TCP tedy představuje spolehlivou komunikaci, která ovšem poskytuje spolehlivý přenos a řídí komunikaci na úkor rychlosti.

4.2.2 UDP (User Datagram Protocol)

UDP je, podobně jako TCP, komunikační protokol určený k navazování vysokorychlostního, chybově tolerantního spojení mezi aplikacemi komunikujícími po síti. UDP běží nad internetovým protokolem IP a je také často označován jako UDP/IP.

UDP umožňuje komunikaci typu process-to-process. TCP umožňuje komunikaci typu host-to-host. Na rozdíl od TCP, UDP nevysílá pakety a neumožňuje spolehlivou komunikaci. UDP rozesílá zprávy zvané datagramy a poskytuje komunikaci stylem největší snahy.

UDP neposkytuje žádnou chybovou ani řídící kontrolu a je považován za protokol nevyužívající spojení, protože nepotřebuje při komunikaci žádný virtuální spoj.

UDP představuje nespolehlivý způsob přenosu dat po síti, který ale umožňuje výrazně rychlejší komunikaci díky neexistující chybové kontrole.

4.2.3 Výběr komunikačního protokolu

Distribuovaný výpočet nad vícevrstvou neuronovou sítí s datovým modelem využívající přístup centralizovaného asynchronního stochastického sestupu gradientu potřebuje ke svému rozumnému a použitelnému fungování rychlou a spolehlivou komunikaci. Z průzkumu v podkapitolách výše je ovšem zřejmé, že tyto dva parametry nejdou ruku v ruce.

Rozhodnutí tedy leží v tom, jestli je pro naši komunikaci důležitější rychlost, či spolehlivost.

Spolehlivá komunikace je při distribuovaném výpočtu nutná, protože těžko se může výpočet posouvat dál, když data nepřichází nebo přichází poškozená. Rychlá komunikace je ovšem také nutností, protože pokud je komunikace pomalá, může dojít k absolutní eliminaci výhod paralelismu při výpočtu.

Spolehlivost se dá do určité míry zaručit i u UDP a to tím, že se komunikace bude provádět opakovaně tak, aby se snížila možnost ztráty dat po cestě k druhé straně a zároveň tak, že komunikace bude probíhat v uzavřené stabilní síti, kde datům po cestě nehrozí příliš mnoho komplikací.

Naopak rychlost se dá trochu zvýšit například kódováním zpráv mezi účastníky komunikace, ale převážně zůstává odpovědnost na bedrech komunikačního protokolu.

Proto bylo učiněno rozhodnutí využít pro prototyp knihovny komunikaci za pomocí protokolu UDP. Je zřejmé, že implementace celé komunikace by zabrala příliš mnoho času a její ladění ještě více.

Proto bude využita UDP komunikační knihovna Aeron.

4.3 Komunikační knihovna Aeron[6]

Aeron je komunikační knihovna využívající UDP a jiné protokoly pro singlecast i Multicast komunikaci od společnosti Real Logic. Určená je pro jazyky Java a C++. Dostupný je také .NET klient, ale je poskytován třetí stranou.

Všichni tři klienty, tedy pro Javu, C++ a .NET, spolu mohou komunikovat velice efektivně jak lokálně, tak po síti.

Pro Aeron je nejdůležitější vlastností výkon. Knihovna je tak navržena se zaměřením na co možná nejvyšší propustnost a s co možná nejmenším zpožděním. Využívá vlastní implementaci jednoduchého binárního kódování pro nejlepší možný výkon během kódování a dekódování.

Aeron je navržený tak, aby běžel nad různými typy přenosových médií, jako jsou sdílené paměti, InfiniBand, UDP, TCP, Raw IP, HTTP, WebSockets, BLE a další. Z tohoto vychází následující předpoklady:

- Přenosové médium může být proudové médium, jako TCP.
- Přenosové médium může podporovat pouze Unicast mód.

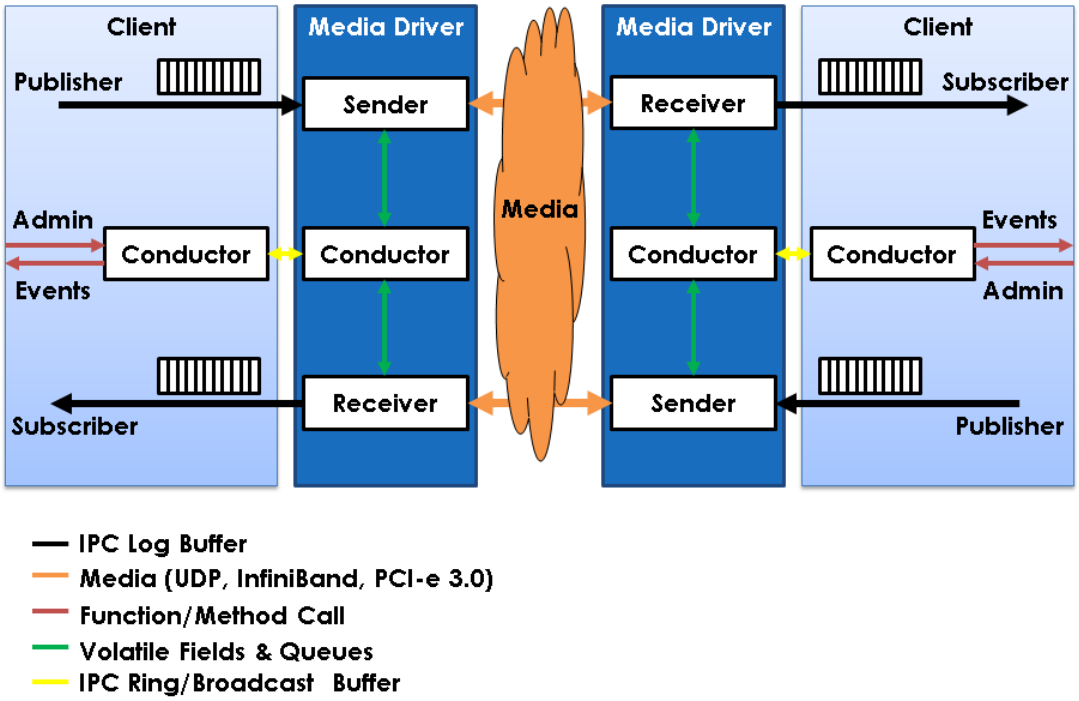
Na Aeron lze pohlížet jako na komunikační protokol a tedy může operovat nad nespolehlivým médiem aby poskytl spolehlivý, na spojení orientovaný proud jako OSI Transport Layer. Z těchto důvodů bylo potřeba vzít v potaz další předpoklady, které Aeron bude detekovat a opravovat. Mezi ty patří:

- Může nastat duplikace paketů.
- Pakety se mohou ztratit.
- Pakety mohou dorazit v nepředpokládaném pořadí.

Aeron také předpokládá určitě operační podmínky. Jednou z nich je fakt, že počet proudů bude nízký. Počet proudů se očekává být nižší než tisíc, někdy dokonce menší než sto.

Aeron je tedy navrhnut k práci ruku v ruce s nižší paralelní datovou strukturou, na které je postaven. Toto vede k efektivní a symbiotické spolupráci mezi použitými datovými strukturami a základovými protokoly.

Architekturu knihovny Aeron si lze prohlédnout na obrázku 8.



Obrázek 8: Architektura knihovny Aeron[6]

4.3.1 Aeron nad UDP

Aeron funguje nad UDP v jednom ze tří módů. Prvním z nich je Unicast, tedy komunikace bod-bod. Druhým je Multicast mód. Posledním módem je Multi-Destination-Cast.

Tyto módy jsou spojeny s jednosměrným datovým tokem. Autoři knihovny Aeron nevyvracejí možnost vytvořit obousměrnou komunikaci pomocí mixování módů. Tedy využít Unicast v jednom směru a Unicast ve směru opačném.

Vymezení kanálu pro UDP komunikaci pomocí Aeron knihovny je založeno na následujícím URI schéma.

1. aeron:udp?[interface=local-interface[:local-port]|]endpoint=receiver-address:receiver-port[|control=explicit-control-address:control-port]

Význam adresy a portu příjemce se může lehce lišit v závislosti na tom, jestli definujeme Unicast nebo Multicast.

4.3.2 Aeron Unicast mód

V Unicast módu příjemce komunikace poslouchá na specifickém rozhraní (definovaném IP adresou) a UDP portu. Dále příjemce posílá zprávy o stavu a NAK zprávy (tedy zprávy o negativním stavu přijetí zprávy) zpátky odesílateli na jeho IP adresu.

Následuje příklad URI pro Unicast mód.

1. aeron:udp?endpoint=10.0.0.1:12345

4.3.3 Aeron Multicast mód

V Multicast módu příjemci a odesílatelé poslouchají a posílají data na specifický interface, danou IP multicast adresu/skupinu a cílový UPD port datový koncového bodu. Tato IP adresa musí být lichá.

Podobně jako datový koncový bod, příjemci a odesílatelé přijímají a posílají data na interface kontrolního koncového bodu definovaný specifickou IP multicast adresou/skupinou a daným UDP portem. Na tomto koncovém bodu se objevují pouze stavové a NAK zprávy. Jeho IP multicast adresa musí být další větší sudá adresa po té od datového kontrolního bodu.

Následuje příklad URI pro Multicast mód.

1. Rozhraní aeron:udp?endpoint=10.0.0.1:4050 představuje datový koncový bod
2. Adresa 10.0.0.2 poté bude rezervována pro kontrolní koncový bod

4.3.4 Aeron Multi-Destionation-Cast operační mód

V Multi-Destionation-Cast módu odesílatel odesílá data přímo seznamu příjemců a spravuje si je jako Multicast skupinu. Využívá ovšem Unicast UDP komunikaci místo Multicast. Seznam příjemců může být řízen manuálně tak, že se příjemci přidávají a odebírají ručně. Dále může být řízen dynamicky, kdy se všichni příjemci přidávají sami a sami se i odebírají, pokud jsou dlouho neaktivní.

V manuálním módu příjemci poslouchají opět na specifickém rozhraní definovaném IP adresou a UPD portem.

Příjemci posílají zpátky stavové a NAK zprávy přímo na odesílatelův kontrolní koncový bod.

Následují příklady URI pro manuální Multi-Destionation-Cast operační mód.

1. aeron:udp?control=10.0.0.1:4050|control-mode=manual představuje rozhraní odesílatelů.
2. aeron:udp?endpoint=10.0.0.1:4051 představuje rozhraní jednoho příjemce.
3. aeron:udp?endpoint=10.0.0.2:4050 představuje rozhraní dalšího příjemce.
4. aeron:udp?endpoint=10.0.0.1:4051 je přidán do seznamu příjemců. Odesílatelé nyní posílají data na příjemci s rozhraním aeron:udp?endpoint=10.0.0.1:4051
5. aeron:udp?endpoint=10.0.0.2:4050 je přidán do seznamu příjemců. Odesílatelé nyní rozesílají data příjemcům aeron:udp?endpoint=10.0.0.2:4050 a aeron:udp?endpoint=10.0.0.1:4051

V dynamickém módu příjemci poslouchají na specifickém rozhraní, IP adrese a UDP portu, a odesílají stavové a NAK zprávy zpátky na IP odesílatele.

Příjemci musí v dynamickém módu bez výjimky odesílat umělé stavové zprávy na kontrolní IP adresu pokud neposílají stavové zprávy na danou IP adresu kvůli standardnímu komunikačnímu ruchu. Tyto umělé zprávy se od standardních odlišují speciálním SETUP návěstím.

Pokud příjemce neodešle na kontrolní IP adresu standardní nebo umělou stavovou zprávu, je po definovaném časovém úseku vyřazen ze seznamu příjemců.

Následují příklady URI pro dynamický Multi-Destionation-Cast operační mód.

1. aeron:udp?control=10.0.0.1:4050 představuje rozhraní odesílatelů.
2. aeron:udp?endpoint=10.0.0.2:4051|control=10.0.0.1:4050 značí, že odesílatelé nyní posílají data příjemci na adrese 10.0.0.2:4051
3. aeron:udp?endpoint=10.0.0.3:4052|control=10.0.0.1:4050 značí, že odesílatelé nyní posílají data příjemcům na adresách 10.0.0.2:4051 a 10.0.0.3:4052

4.3.5 Výběr metody komunikace z knihovny Aeron

Pro efektivní fungování komunikace mezi jednotlivými výpočetními uzly distribuovaného výpočtu vícevrstvé neuronové sítě bude zapotřebí vytvořit komunikaci typu Master-Slave. Toto rozhodnutí jasně vychází z potřeby použité metody datového paralelismu centralizovaného asynchronního stochastického sestupu gradientu.

V takovém modelu je potřeba mít jeden hlavní centrální prvek. V komunikační terminologii se jedná o Master prvek, který drží hlavní a vždy aktuální kopii dat a také rozesílá učící data jednotlivým kopiím vícevrstvé neuronové sítě.

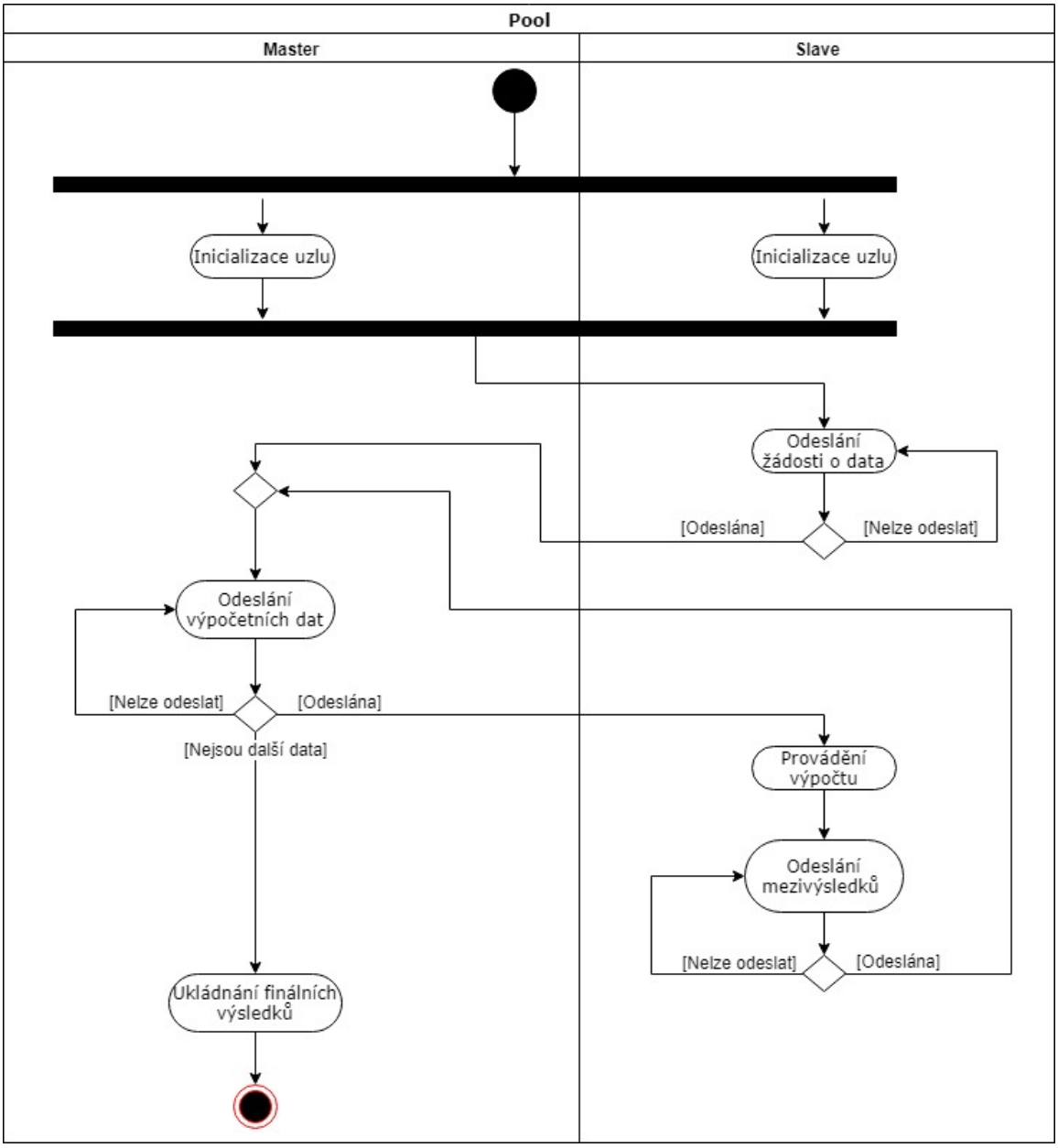
Každý výpočetní uzel udržující svou kopii neuronové sítě pak funguje jako příjemce a zpracovatel dat z Master uzlu, zde mluvíme o Slave uzlu. Právě proto se jedná o komunikaci typu Master-Slave.

Slave uzel se ovšem nemůže chovat pouze jako pasivní účastník komunikace. Protože zpracovává data a vytváří mezivýsledky, je nutné, aby závěry svého výpočtu distribuoval zpátky na Master uzel, neboť Master uzel si musí udržovat nejaktuálnější podobu vícevrstvé neuronové sítě a dále rozesílat tuto podobu Slave uzlům. Nakonec bude Master uzel držet výslednou naučenou podobu neuronové sítě a tu uchová jako výsledek výpočtu.

Celý proces si lze prohlédnout na obrázku 9.

Komunikace není z principu příliš složitá, ale bude mít samozřejmě velké nároky na rychlost. Komunikace počítá s tím, že se během výměny dat budou ztrácet celé zprávy a je připravena výměnu zpráv opakovat. Z diagramu aktivit lze vidět, že Master uzel i Slave uzel budou oba vystupovat jako příjemci i odesílatelé.

Z těchto poznatků je zřejmé, že Slave uzel bude komunikovat s Master uzlem pomocí Unicast módu. Jedná se o přímou komunikaci mezi pouze dvěma účastníky a není tedy nutné zvažovat Multicast v jakékoliv podobě.



Obrázek 9: Aktivitní diagram komunikace mezi uzly Master a Slave

Master uzel naopak bude komunikovat s různým množstvím Slave uzlů a bude tedy využívat některý z módů pro multicast komunikaci. Nabízí se dva módy, Multicast mód a Multi-Destionation-Cast mód.

Z pohledu návrhu prototypu knihovny vychází Multicast mód i Multi-Destionation-Cast mód obdobně výhodně a oba nabízejí potřebnou funkcionalitu. Multi-Destionation-Cast mód oproti módu Multicast nabízí jednu značnou výhodu. Tou je možnost spravovat přihlášené příjemce, a proto byl pro prototyp knihovny vybrán právě tento mód. Bude použit v manuálním režimu, aby byla zachována co možná nejvyšší kontrola nad chováním komunikace a její správou. Takto bude

rozhodnutí o odpojení příjemce zcela v rukou algoritmu řídícího komunikaci a ne v závislosti na nastavení časového limitu v Aeron knihovně.

4.4 Návrh informačních a řídících stavů komunikace

Celá komunikace mezi jednotlivými výpočetními uzly, stejně jako mezi centrálním uzlem a zbytkem uzlů, bude probíhat za použití nespolehlivé komunikace pomocí protokolu UDP. Pro existenci takové komunikace je naprosto nezbytné, aby byla nějakým způsobem řízena.

Přímou kontrolu nad komunikací budou mít nástroje poskytované knihovnou Aeron. Je ale také potřeba mít i další vrstvu kontroly, která prototypu knihovny dovolí uchovávat si přehled a nutné informace o průběhu komunikace. Je potřeba identifikovat v jakém stavu se výpočetní uzly nacházejí a neustále mít pod kontrolou stav celého výpočtu, jeho uzlů i dostupných dat.

Abychom mohli takovou komunikaci jednoduše a přehledně řídit, je potřeba si vytvořit systém stavů, které budou popisovat situaci, ve které se komunikace a jednotlivé uzly právě nacházejí. Je také potřeba identifikovat možné stavy, ve kterých se komunikace nebo jednotlivé uzly můžu nacházet, popsat je a postarat se o to, aby byly správně zachyceny a ošetřeny.

Pokud by se objevil stav, který není v knihovně popsaný a knihovna by neměla dostatek informací na to, aby si s ním dokázala poradit, mohlo by to vést k úplnému kolapsu komunikace i výpočtu samotného. A tedy zmaření klidně i několika hodin výpočetního času.

V následujícím výčtu jsou zachyceny všechny stavy, které byly identifikovány při návrhu jako možné a s jejichž nastáním musí prototyp knihovny počítat.

1. Stav NEINICIALIZOVÁN
2. Stav ODESLAT
3. Stav NEODESLAT
4. Stav UKONČEN

Všechny tyto stavy jsou nějak spojeny s komunikací mezi výpočetními uzly a řídícím uzlem, a zároveň hrají významnou roli při řízení chování samotných uzlů. O tom, v jakém stavu se daný výpočetní uzel nachází, ví pouze řídící uzel. Tento uzel si uchovává informaci o každém výpočetním uzlu. Výpočetní uzly identifikuje pomocí jejich rozhraní, tedy kombinace IP adresy a portu.

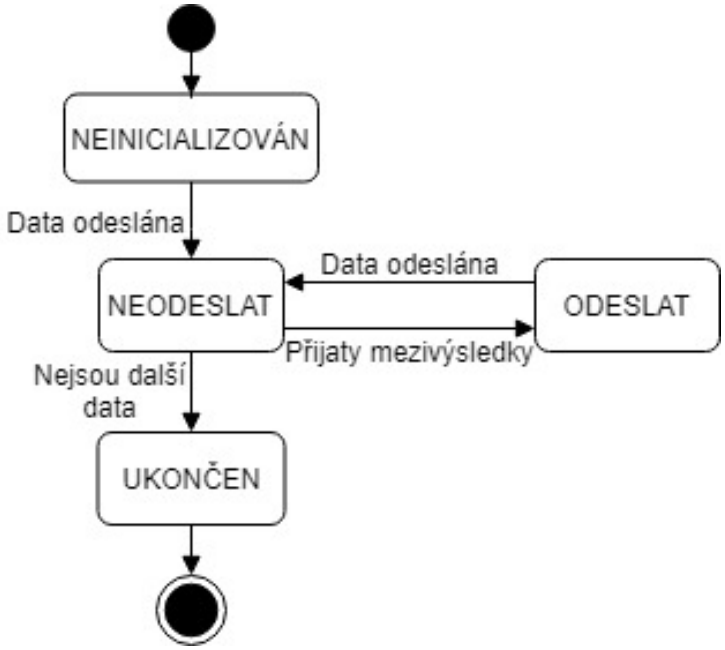
Následuje popis jednotlivých stavů a jejich význam.

Stavy a přechody mezi nimi lze přehledně vidět na obrázky 10.

4.4.1 Stav NEINICIALIZOVÁN

Stav NEINICIALIZOVÁN je jeden z nejjednodušších stavů, které prototyp knihovny využívá.

V tomto stavu se nacházejí všechny výpočetní uzly po startu řídícího centrálního uzlu, tedy ve chvíli, kdy je řídící uzel inicializován. Pokud se některý z výpočetních uzlů nachází ve stavu



Obrázek 10: Stavový diagram komunikace

NEINICIALIZOVÁN, znamená to, že daný uzel nikdy neprovedl žádnou komunikaci s řídícím uzlem. Tento stav signalizuje problém v komunikaci mezi výpočetním uzlem a řídícím uzlem nebo problém s inicializací výpočetního uzlu.

Stav NEINICIALIZOVÁN působí v aplikaci jako pojistka, která zajišťuje, že se už od začátku výpočtu nefungující uzly budou ignorovat a řídící uzel bude vědět, že na ně nemá brát ohled.

4.4.2 Stav ODESLAT

Stav ODESLAT funguje v kooperaci se stavem NEODESLAT. Slouží k signalizaci toho, že daný výpočetní uzel se nachází v stavu čekání na data a řídící uzel mu musí potřebná učící data dodat. Pokusy o odeslání dat výpočetnímu uzlu budou trvat tak dlouho, dokud se nepodaří data uzlu dodat.

Do stavu ODESLAT se uzel může dostat pouze ze stavu NEODESLAT.

4.4.3 Stav NEODESLAT

Jak již bylo zmíněno výše, stav NEODESLAT funguje v kooperaci se stavem ODESLAT. Stav NEODESLAT signalizuje, že výpočetní uzel právě provádí výpočet a není v jeho případě potřeba žádná činnost ze strany řídícího uzlu. Pro všechny výpočetní uzly ve stavu NEODESLAT řídící uzel sleduje své rozhraní pro příjem zpráv a očekává příchod mezivýsledků.

Do stavu NEODESLAT se lze dostat ze stavů ODESLAT a NEINICIALIZOVÁN.

4.4.4 Stav UKONČEN

Posledním stavem je stav UKONČEN. Stav UKONČEN značí, že daný výpočetní uzel ukončil kompletně svou činnost a nelze ho už nadále kontaktovat. Stav UKONČEN slouží k signalizaci celkového ukončení distribuovaného výpočtu.

Do stavu UKONČEN se dostává výpočetní uzel tak, že mu řídící uzel místo dat zašle ukončovací signál a čeká, až přestane výpočetní uzel odpovídat na jeho komunikačním rozhraní. Při rozhodování o ukončení programu musí být všechny uzly buď ve stavu UKONČEN nebo NEINICIALIZOVÁN.

Do stavu UKONČEN se lze dostat ze stavu ODESLAT.

4.5 Vícevrstvá neuronová síť typu Backpropagation

Součástí návrhu prototypu knihovny pro paralelizaci vícevrstvých neuronových sítí je návrh samotné neuronové sítě.

Ačkoliv již existující aplikace Modeler neuronových sítí v sobě implementaci vícevrstvé neuronové sítě má a jedná se o síť, se kterou bude muset prototyp knihovny počítat a pracovat, tak se bohužel jedná o síť nevhodnou pro testování vlastností prototypu knihovny.

Aby bylo možné knihovnu otestovat a správně navrhnout, je potřeba vytvořit plnohodnotnou funkční implementaci vícevrstvé neuronové sítě typu Backpropagation.

Bude potřeba implementovat vícevrstvou neuronovou síť využívající při učení algoritmus známý jako Backpropagation. Bude nutné ji vytvořit tak, aby bylo možné její chod paralelizovat nad více jádry CPU či GPU. Tato síť musí být volně konfigurovatelná, aby bylo možné jednoduše měnit počet vrstev v síti, počet neuronů v daných vrstvách a také parametry sítě.

4.5.1 Neuronová síť

Umělá neuronová síť představuje počítačovou reprezentaci biologické neuronové sítě. Vytvoření umělé neuronové sítě není nic jiného, než snaha vytvořit umělou variantu nervového systému.

Nervový systém zvířat se skládá z buněk zvaných neurony a jejich vzájemných spojení. Každý živý tvor má ve svém těle tisíce až miliony těchto propojených buněk. Pokud se na jejich fungování podíváme více z nadhledu, můžeme říci, že fungují a vytvářejí spolu jakousi formu řízení skrz komunikaci. Každá z nich má složitou strukturu určující, jak se naloží s různými druhy signálů, které buňka zachytí. Každou z těchto buněk si lze představit jako elektrické logické hradlo.

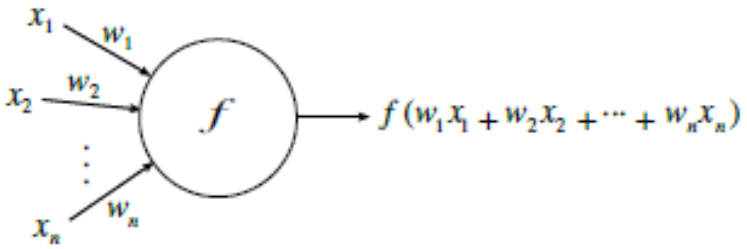
Neuron je ale mnohem pomalejší než elektrické logické hradlo. Zatímco hradlo dokáže přepnout svou polohu v pár nanosekundách, neuron potřebuje pro svou reakci několik milisekund. Přesto je lidský mozek schopen řešit úlohy, které dneska ještě žádný počítač efektivně nedovede.

Je tedy zřejmé, že pokud bychom dokázali vytvořit a učit umělou neuronovou síť, je možné řešit úlohy, na které nám dosavadní možnosti standardních algoritmů nestačí.

Návrh a implementace neuronové sítě vychází z její biologické předlohy. Jako základ sítě se používá neuron, který reprezentuje základní stavební kámen sítě. Každý neuron má N vstupních spojení, dendritů a jedno výstupní spojení, axon.

Zjednodušeně řečeno funguje každý neuron v umělé síti tak, že převezme signál ze svých N vstupů, aplikuje jej na nějakou primitivní funkci a výsledek pošle na výstupní spojení. Síla signálu při převodu na další neuron je ovlivněna takzvanou váhou spoje. Tato hodnota určuje, s jakou silou se bude signál propagovat na další napojené neurony.

Příklad abstraktního umělého neuronu lze vidět na obrázku 11.



Obrázek 11: Abstraktní umělý neuron[7]

Pokud se budeme dívat na každý umělý neuron v neuronové síti jako na primitivní funkci schopnou transformovat svůj vstup na přesně definovaný výstup, pak umělá neuronová síť není nic jiného než síť primitivních funkcí. Různé modely umělé neuronové sítě se liší převážně v předpokladech o těchto primitivních funkcích, ve vzoru použitém při spojování neuronů a časování přenosu informací.

Učení neuronové sítě pak nepředstavuje nic jiného, než upravování vah jednotlivých spojů mezi neurony, a tak ovlivňování toho, jak silně budou následující neurony reagovat na signál neuronu, na který jsou napojeny.

4.5.2 Dopředná propagace

Dopředná propagace je přístup k propagování vstupních hodnot skrz síť dopředným směrem (což naznačuje už název přístupu).

Hlavní cílem je vzít vstup na vstupní vrstvě neuronové sítě x a použít ho k výpočtu dané funkce, která je pak využita k výpočtu výstupu y. Dále se použije výstup předchozí vrstvy jako vstup pro vrstvu následující a provádí se stejné operace. Proto celý přístup nazýváme dopředná propagace.

Sítě využívající dopřednou propagaci lze reprezentovat jako kompozici mnoha různých funkcí. Každý model je spojený s acyklickým grafem popisujícím, jak jsou funkce spojeny dohromady. Například můžeme mít tři funkce f_1, f_2, f_3 spojené do řady za sebou. Abychom mohli spočíst

$f(x)$, musíme ji zformovat takto $f(x) = f_1(f_2(f_3(x)))$. Pak f_1 představuje první vrstvu, f_2 představuje druhou vrstvu a f_3 představuje třetí vrstvu.

Vrstvy mezi první vstupní vrstvou a poslední výstupní vrstvou nazýváme skryté vrstvy, protože data pro učení neuronové sítě neukazují požadovaný výstup těchto vrstev. Vícevrstvá neuronová síť může obsahovat různé množství těchto skrytých vrstev a ty mohou obsahovat různé množství skrytých jednotek. Skrytá jednotka v podstatě představuje neuron, který bere na vstupu hodnoty z jednotek předchozích vrstev a počítá svou vlastní výstupní hodnotu.

Neuronové sítě s dopřednou propagací mají výhodu oproti lineárnímu strojovému učení v tom, že lineární strojové učení je limitováno pouze na lineární funkce, zatímco neuronové sítě takto limitované nejsou. Pokud data nejsou lineární, má lineární strojové učení problém s aproximací, ale pro neuronové sítě tento úkol není problém. Skryté vrstvy jsou použity pro snížení linearity a mění reprezentaci dat pro lepší generalizací nad funkcí.

4.5.3 Backpropagation

S vzrůstající komplexností, přibývajícími parametry a komplikovanějšími topologiemi vícevrstvých neuronových sítí roste i složitost hledání správné kombinace úprav hodnot vah. Jednou z populárních metod, které tento problém řeší, je právě algoritmus Backpropagation.

Algoritmus Backpropagation hledá minimální chybovou funkci v prostoru vah pomocí metody sestupného gradientu. Kombinace vah, která minimalizuje chybovou funkci, je považována za řešení učení neuronové sítě. Kvůli algoritmu musí být zaručena kontinuita a odlišitelnost chybové funkce, protože tato metoda vyžaduje výpočet gradientu chybové funkce při každém kroku iterace.

Jednou z nejpopulárnější aktivačních funkcí pro algoritmus Backpropagation je funkce sigmoid s_c. Jedná se o reálnou funkci definovanou následovně:

$$s_c(x) = \frac{1}{1 + e^{-cx}} \tag{3}$$

Konstanta c může být zvolena dle libosti a $\frac{1}{c}$ se nazývá teplotní parametr v stochastických neuronových sítích.

4.5.4 Algoritmus Backpropagation

Lze definovat kompletní Backpropagation algoritmus, který bude fungovat s libovolnými sítěmi typu dopředné propagace s různými aktivačními funkcemi na jejích neuronech. Tento algoritmus předpokládá, že pracuje s neuronovou sítí, která má právě jeden vstupní neuron a jeden výstupní neuron.

Mějme neuronovou síť s jedním reálným vstupem x a síťovou funkcí F. Derivace funkce $F'(x)$ lze vypočítat ve dvou fázích:

- Dopředná propagace: vstupní hodnota x je vpuštěna do neuronové sítě. Primitivní funkce v jednotlivých neuronech a jejich derivace jsou spočteny na každém neuronu. Derivace jsou uchovány.

- Backpropagation: konstanta 1 je vložena do výstupní jednotky a neuronová síť propaguje zpět. Příchozí informace do neuronu je přičtena a výsledek je multiplikován hodnotou uloženou v neuronu. Výsledek sesbíraný na vstupním neuronu je derivací síťové funkce vzhledem k x.

4.5.5 Učení pomocí Backpropagation

Berme v potaz učící problém neuronových sítí. Cílem je snížit chybovou funkci E, která záleží na vahách neuronové sítě. Musí tedy řešit všechny váhy sítě jednu po druhé. Kroky dopředné propagace jsou provedeny standardně, ale musí si v sobě každý jeden neuron uchovat svou výstupní hodnotu.

Pak je v neuronové sítí proveden krok algoritmu Backpropagation, který spočítá chybovou funkci a následně se zaměří na jednu z vah $w_{i,j}$ jejíž spojení vede z neuronu i do neuronu j. Tato váha může být brána jako vstupní brána do podsítě začínající $w_{i,j}$ a končící ve výstupní jednotce. Informace, která vstoupila do podsítě v dopředné propagaci je $o_i w_{ij}$, kde o_i je uložena jako výstup neuronu i. Krok algoritmu Backpropagation spočte gradient E vzhledem k vstupu, tedy $\frac{\partial E}{\partial o_i w_{ij}}$. Protože v kroku Backpropagation je o_i chápána jako konstanta, lze definovat následující vztah.

$$\frac{\partial E}{\partial w_{ij}} = o_i \frac{\partial E}{\partial o_i w_{ij}} \tag{4}$$

Shrnuto, krok algoritmu Backpropagation je vykonán standardně. Všechny podsítě, definované každou jednou váhou neuronové sítě, můžou být řešeny současně, ale je nutné v každém neuronu i uchovávat následující:

- výstup o_i neuronu v kroku dopředné propagace.

- souhrný výsledek zpětného chodu v kroku Backpropagation až po tento neuron. Tato hodnota se nazývá chybou Backpropagation.

Když je chyba Backpropagation na neuronu j označena jako δ_j, tak je možné vyjádřit parciální derivaci E vzhledem k w_{ij} jako následující.

$$\frac{\partial E}{\partial w_{ij}} = o_i \delta_j \tag{5}$$

Ve chvíli, kdy se spočtou všechny parciální derivace, je možné vykonat sestup gradientu připočtením ke každé jednotlivé váze w_{ij} následující inkrement.

$$\Delta w_{ij} = \gamma o_i \delta_j \tag{6}$$

Tento korekční krok je nutný pro transformaci algoritmu backpropagation do metody učení pro neuronové sítě.

Výše popsaný princip je uplatnitelný pro všechny možné topologie využívající dopřednou propagaci. Z toho přístupu lze snadno odvodit i požadavky na hardwarovou implementaci potřebnou pro algoritmus Backpropagation.

4.5.6 Algoritmus Backpropagation pro vícevrstvé neuronové sítě

Velmi důležitým případem sítí s dopřednou propagací jsou ty, které mají jednu nebo více skrytých vrstev.

Pro tuto část textu uvažujeme síť, která má n vstupních neuronů, k skrytých neuronů a m výstupních neuronů. Jako aktivační funkci budeme používat funkci sigmoid.

Poté, co se zvolí hodnoty vah vícevrstvé neuronové sítě náhodně, je backpropagation algoritmus použit k výpočtu nezbytných korekcí hodnot vah. Algoritmus může být rozdělen do následujících kroků:

1. Výpočet dopředné propagace.
2. Backpropagation na výstupní vrstvě.
3. Backpropagation na skryté vrstvě.
4. Úprava hodnot vah.

Algoritmus se ukončí ve chvíli, kdy hodnota chybové funkce je přijatelně malá nebo dojde k průchodu potřebným množstvím epoch.

První krok: výpočet dopředné propagace

První krok algoritmu je výpočet dopředné propagace. Neuronové síti je předložen vektor o, který představuje vstupní hodnoty sítě. Neurony si uloží hodnoty spočtené během dopředné propagace. Výsledky derivace aktivační funkce jsou také uloženy v jednotlivých neuronech.

Druhý krok: backpropagation na výstupní vrstvě

V druhém kroku neuronová sít provede backpropagation na výstupní vrstvě.

V tomto kroku řešíme první část parciálních derivací $\frac{\partial E}{\partial w_{ij}}$. Nyní je získána chyba backpropagation pro tuto vrstvu ∂_j. Tedy je stanoveno:

$$\delta_j = o_j(1 - o_j)(o_j - t_j) \tag{7}$$

A tedy potřebná parciální derivace:

$$\frac{\partial E}{\partial w_{ij}} = [o_j(1 - o_j)(o_j - t_j)]o_i = \delta_j o_i \tag{8}$$

Pro vykonání tohoto kroku je w_{ij} považováno za proměnnou a její vstup o_i za konstantu.

Shrnuto, máme na vstupní straně váhy w_{ij} máme vektor o_i a na výstupní straně máme chybu backpropagation δ_j.

Třetí krok: backpropagation na skryté vrstvě

Třetí krok představuje provedení backpropagation na skryté vrstvě.

Je potřeba opět spočíst parciální derivaci $\frac{\partial E}{\partial w_{ij}}$. Každý neuron j v skryté vrstvě je spojený s každým neuronem ve výstupní q vrstvě vahou w_{jq}, kdy $q = 1, \ldots, m$. Chyba backpropagation až po neuron j musí být ve skryté vrstvě spočtena s ohledem na všechny možné zpětné cesty. Chyba backpropagation je tedy stanovena takto:

$$\delta_j = o_j(1 - o_j) \sum_{q=1}^{m} w_{jq}\delta_q \tag{9}$$

A tedy parciální derivace bude vypadat:

$$\frac{\partial E}{\partial w_{ij}} = \delta_j o_i \tag{10}$$

Chybu backpropagation lze spočítat stejně pro jakékoliv množství skrytých vrstev a vztah pro parciální derivaci E si uchovává stejnou analytickou formu.

Čtvrtý krok: úprava vah

Čtvrtým a posledním krokem je úprava hodnot vah.

Po spočtení všech parciálních derivací jsou váhy neuronové sítě upraveny pomocí gradientu. Učící konstanta γ definuje délku kroku korekce. Korekce vah jsou definovány tímto vztahem.

$$\Delta w_{ij} = -\gamma o_i \delta_j, \; pro \; i = 1, \ldots, k+1; j = 1, \ldots, m \tag{11}$$

Je zde využit fakt, že $o_{k+1} = 1$.

Je velmi důležité upravovat váhy chyby až poté, co se spočte chyba backpropagation pro všechny neurony v sítí. Jinak se korekce hodnot vah prováže s výpočtem chyby backpropagation a úpravy nadále nekorespondují se směrem gradientu.

4.5.7 Učení s více než jedním vzorem k učení

V případě, že máme více než jeden učící vzor p, tedy $p > 1$, je k výpočtu chybové funkce pro každý ze vzorů použita rozšířená neuronová síť. Úpravy vah jsou vypočítány pro každý vzor jednotlivě a tedy pro váhu w_{ij} následující korekce:

$$\Delta_1 w_{ij}, \Delta_2 w_{ij}, \ldots, \Delta_p w_{ij} \tag{12}$$

Potřebná úprava vah pomocí grandientu je tedy následující:

$$\Delta w_{ij} = \Delta_1 w_{ij} + \Delta_2 w_{ij} + \cdots + \Delta_p w_{ij} \tag{13}$$

V tomto případě mluvíme o dávkové nebo Off-line úpravě. Často jsou ale úpravy provedeny sekvenčně po každém jednom vzoru, a pak mluvíme o Online úpravě. V případě Online úprav korekce nesledují přímo směr gradientu, ale jestli je učící vzor vybrán náhodně, tak směr osciluje okolo přesného směru gradientu a algoritmus využívá formu sestupu gradientu.

Pokud se při použití Online úprav přidá do směru gradientu rušení, pomůže to vyhnout se pádu do lokálního minima chybové funkce. Také, pokud množství učících vzorů p dosáhne vysokých čísel, je velmi výpočetně náročné počítat směr gradientu pro každou jednu epochu, která se skládá z mnoha dopředných propagací, a tedy Online úpravy se ukazují jako mnohem efektivnější varianta.

4.5.8 Volba neuronové sítě

Pro úspěšnou implementaci a testování prototypu knihovny pro distribuovaný výpočet na vícevrstvými neuronovými sítě je potřeba mít testovací neuronovou síť. Síť, která vznikne pro potřebu testování, bude parametrizovaná vícevrstvá neuronová síť využívající učení pomocí algoritmu Backpropagation.

Celá síť musí být dynamická, a tedy bude přejímat své parametry z konfiguračního souboru, kde budou definovány všechny podstatné parametry sítě a také je bude jednoduché měnit bez nutnosti zasahovat do implementace sítě.

Celá síť bude popsána pomocí záznamů v konfiguračním souboru. Pro tento účel byl vybrán soubor typu YAML.

Konfigurační soubor bude obsahovat následující parametry:

- počet epoch,
- učící konstanta,
- lambda,
- schéma neuronové sítě.

Parametr Počet epoch představuje počet opakování učení, která se celkově vykonají. Parametr bude zadán přirozeným číslem.

Parametr Učící konstanta představuje učící konstantu, která se předá použité neuronové síti. Ta jej interně využívá pro učící proces. Hodnota parametru Učící konstanta bude zadána reálným číslem v rozsahu (0,1>.

Parametr Lambda také představuje interní parametr lambda neuronové sítě. Je zadán reálným číslem v rozsahu (0,1>.

Poslední parametr je Schéma neuronové sítě. Tento parametr představuje rozložení vrstev neuronové sítě a počtu neuronů v nich. Parametr bude zadán jako posloupnost hodnot, kdy počet hodnot reprezentuje počet vrstev neuronové sítě a jednotlivá přirozená čísla reprezentují počet neuronů v dané vrstvě. Posloupnost musí mít vždy alespoň velikost rovnou dvěma a žádné z čísel nesmí být nulové.

Konfigurační soubory řídícího prvku a výpočetních prvků se budou lehce lišit. Konfigurační soubor řídícího prvku bude obsahovat parametry Počet epoch a Schéma neuronové sítě. Výpočetní prvek pak bude potřebovat parametry Učící konstanta, Lambda a Schéma neuronové sítě.

Schéma neuronové sítě musí být pro řídící prvek i výpočetní prvky vždy stejné.

5 Implementace

Tato část diplomové práce se zabývá implementací prototypu knihovny dle návrhu z předchozí kapitoly. Řeší detaily implementace a také potíže, které během implementace nastaly.

5.1 Část řídící a výpočetní uzel

Řídící i výpočetní uzel představují typické aplikace napsané v jazyce Java. Obě části využívají nástroj Apache Maven pro správu, řízení a automatizaci sestavování. Pomocí nástroje Maven se části sestavují do spustitelných souborů typu JAR.

Části využívají Maven Shady plugin pro vytvoření spustitelného souboru, který umožňuje vytvoření takzvaného tlustého JAR souboru. Tlustý JAR soubor v sobě obsahuje všechny části kódu, konfigurační informace a zároveň i všechny knihovny potřebné pro běh Javovských aplikací.

Části se sestavují jako aplikace pro stolní počítače. Pro spuštění částí v lokálním prostředí je tedy nutné mít ve svém prostředí nainstalováno JRE ve verzi 11.0.1.

Části lze konfigurovat dvěma způsoby.

Prvním z nich je možnost upravit parametry v souborech YAML, které se nachází ve složce *resources*.

Druhou možností je upravení proměnných v konfiguračním souboru *pom* nástroje Maven typ XML, který se nachází v kořenovém adresáři projektu.

Třetí možností je předat konfiguraci za běhu v podobě Java argumentů. Předané parametry v podobě argumentů mají vyšší prioritu než parametry předané pomocí konfiguračních souborů.

Obě aplikace obsahují více Maven profilů, určených pro běh v různých prostředích za rozdílných podmínek. Tyto profily jsou tři - *dev*, *test* a *prod*. Aplikace řídícího i výpočetního uzlu mají totožné profily.

5.1.1 Profil *dev*

Profil *dev* představuje v částech vývojářský profil. Jeho použití je nastaveno jako výchozí možnost při kompilaci nástrojem Maven bez specifikování profilu.

Při použití profilu *dev* se pro běh řídícího i výpočetního uzlu použijí IP adresy lokálního prostředí a jsou jim přiřazeny porty definované v konfiguračním souboru *pom*.

5.1.2 Profil *test*

Profil *test* slouží jako testovací profil, který se využívá pouze ve fázi testování vlastností řídícího i výpočetního uzlu. Tento profil není určen k upravování a je ponechán pouze z důvodů možnosti testovat úpravy na prototypu knihovny po jejím dokončení.

Konfigurace profilu *test* je podobná konfiguraci profilu *dev*, jenomže zde se musí specifikovat nejen port, ale i IP adresy, na kterých budou aplikace poslouchat.

5.1.3 Profil *prod*

Profil *prod* představuje nastavení částí pro běh v produkční prostředí. Nastavení je totožné s nastavením profilu *test*. To je dáno předpokladem, že prostředí určené k testování bude mít identické nastavení jako prostředí produkční.

Parametry profilu *prod* jsou opět IP adresy a porty poslouchajících uzlů.

5.2 Implementace rozhraní pro vlastní neuronové sítě

Aby mohli uživatelé, v tomto případě Modeler neuronových sítí, využívat prototyp knihovny, musí jejich implementace neuronových sítí splnit určité předpoklady, které na ně klade prototyp knihovny.

Nejsnadnější způsob jak v jazyce Java předepsat požadavky na implementaci je pomocí rozhraní. Toto rozhraní se nazývá *NeuralNetController*.

Rozhraní *NeuralNetController* lze vidět v ukázce kódu 1.

```
public interface NeuralNetController {

    void startTraining(TrainingData trainingData);

    List<List<Double>> getGradients();
}
```

Výpis 1: Rozhraní *NeuralNetController*

Rozhraní *NeuralNetController* je standardní rozhraní napsané v jazyce Java. Formátování v tomto textu je upravenou pro lepší čitelnost.

Rozhraní definuje pouze dvě signatury metod.

Signatura metody *startTraining* slouží jako předpis pro metodu, která bude sloužit pro začátek učení vícevrstvé neuronové sítě. Jako parametr přijímá objekt typu *TrainingData*, který v sobě udržuje data nezbytné pro učení neuronové sítě.

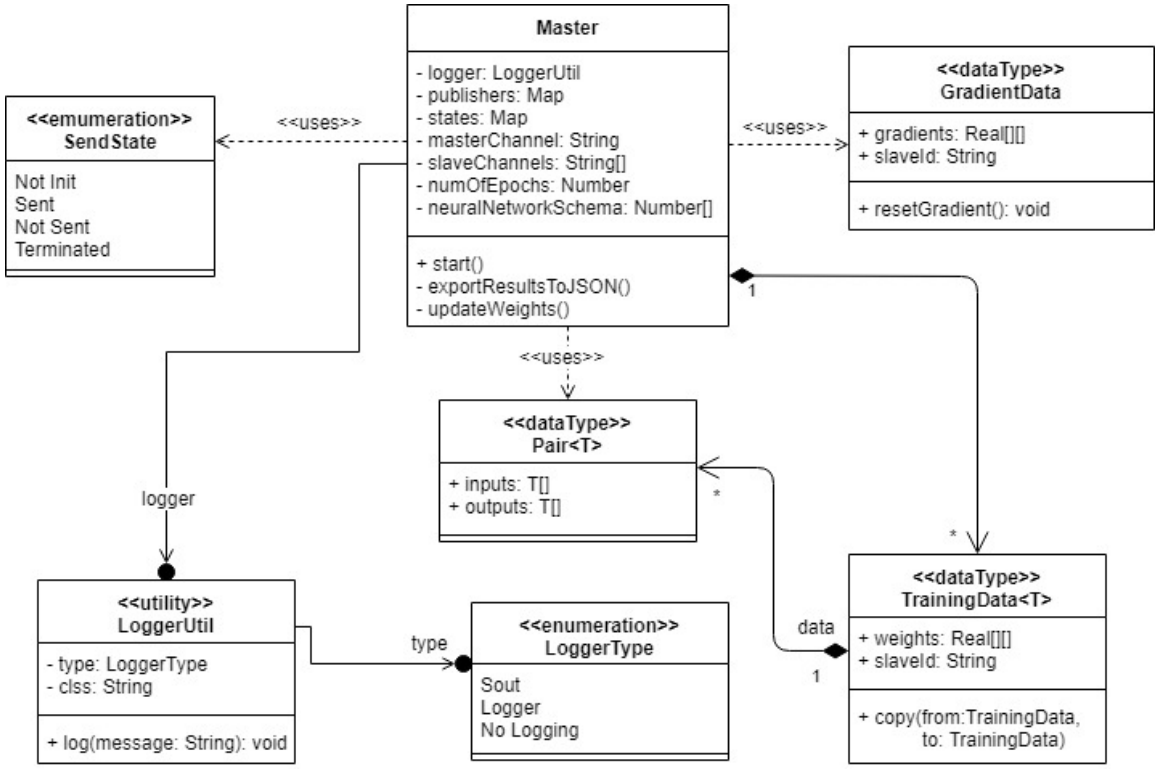
Signatura metody *getGradients* nutí vývojáře implementovat metodu, která definuje přístup k získávání gradientu z učící se sítě. Tato metoda je nutná pro možnost rozesílání mezivýsledků řídícímu prvku.

Metoda vrací seznam seznamů gradientů kvůli možnosti využít funkce, které třída List v jazyce Java poskytuje.

5.3 Implementace tříd řídícího uzlu

Řídící uzel, v implementaci nazývaný *Master*, má poměrně jednoduchou strukturu tvořenou pěti třídami. Kromě těchto tříd obsahuje také tři další třídy potřebné pro načtení konfiguračních informací ze souborů typu YAML.

Třídy řídícího uzlu lze vidět na obrázku 12.



Obrázek 12: Třídní diagram řídícího uzlu

5.3.1 Třída *Master*

Hlavní třídou řídícího uzlu je třída *Master*. Tato třída definuje spouštěcí metodu celé aplikace *start*. Metoda *start* inicializuje ovladače poskytnuté knihovnou Aeron, načte adresy výpočetních uzlů z konfiguračního souboru a inicializuje stav komunikace do úvodního stavu.

Poté ve smyčce kontroluje stav komunikace s výpočetními uzly a dodává jim potřebná data. Zpracovává také příchozí informace a udržuje aktuální podobu neuronové sítě.

Při ukončení komunikace se všemi výpočetními uzly pak vygeneruje výstup a ukončí se.

Třída *Master* využívá vícero datových tříd jako pomocníky nesení dat.

5.3.2 Datová třída *TrainingData*

Třída *TrainingData* slouží jako nositel učících dat a aktuálních hodnot dat. Tato třída se využívá při komunikaci směrem z řídícího prvku do prvků výpočetních. Krom data a hodnot vah obsahuje třída *TrainingData* také proměnnou *slaveId*, ve které je uložený identifikátor příjemce zprávy. Proměnnou *slaveId* využívá výpočetní uzel ke kontrole, zda-li byla příchozí zpráva opravdu určena přímo mu.

5.3.3 Datová třída *GradientData*

Další z tříd, které využívá třída *Master*, je *GradientData*. Na třídu *GradientData* lze pohlížet jako na opak třídy *TrainingData*.

Třída *GradientData* slouží jako datový nosič v komunikaci mezi výpočetním a řídícím uzel. Řídící uzel vystupuje v roli příjemce a výpočetní uzel v roli odesílatele. Tato třída v sobě obsahuje pouze kolekci gradientů v proměnné *gradients* a také identifikátor odesílatele zprávy v proměnné *slaveId*.

Řídící uzel následně identifikuje odesílatele dat, upraví status komunikace a upraví svou neuronovou síť pomocí nově dodaných dat.

5.3.4 Pomocná třída *LoggerUtil*

Poslední třídou, která stojí za zmínku, je třída *LoggerUtil*. Jedná se o pomocnou třídou určenou jako obal pro informační výpisy z aplikace. Pomocí jednoduchého nastavení, kde třída přijímá v konstruktoru parametry typ výstupu a třídu provádějící výpis.

Následně pak kdekoliv v textu stačí zavolat metodu *log* s textovou zprávou jako parametrem a třída se o její výpis postará sama. Ve výpisu je vidět text zprávy a informace o tom, která třída tento výpis vytvořila.

5.4 Implementace tříd výpočetního uzlu

Aplikace výpočetního uzlu, která se v kódu nazývá *Slave*, představuje přímou strukturu sedmi tříd, které využívá k řízení výpočtu. Kromě těchto sedm tříd obsahuje ještě tři třídy potřebné k načtení konfigurace.

Třídní diagram výpočetního uzlu lze vidět na obrázku 13.

5.4.1 Třída *Slave*

Hlavní třídou výpočetního uzlu je třída *Slave*.

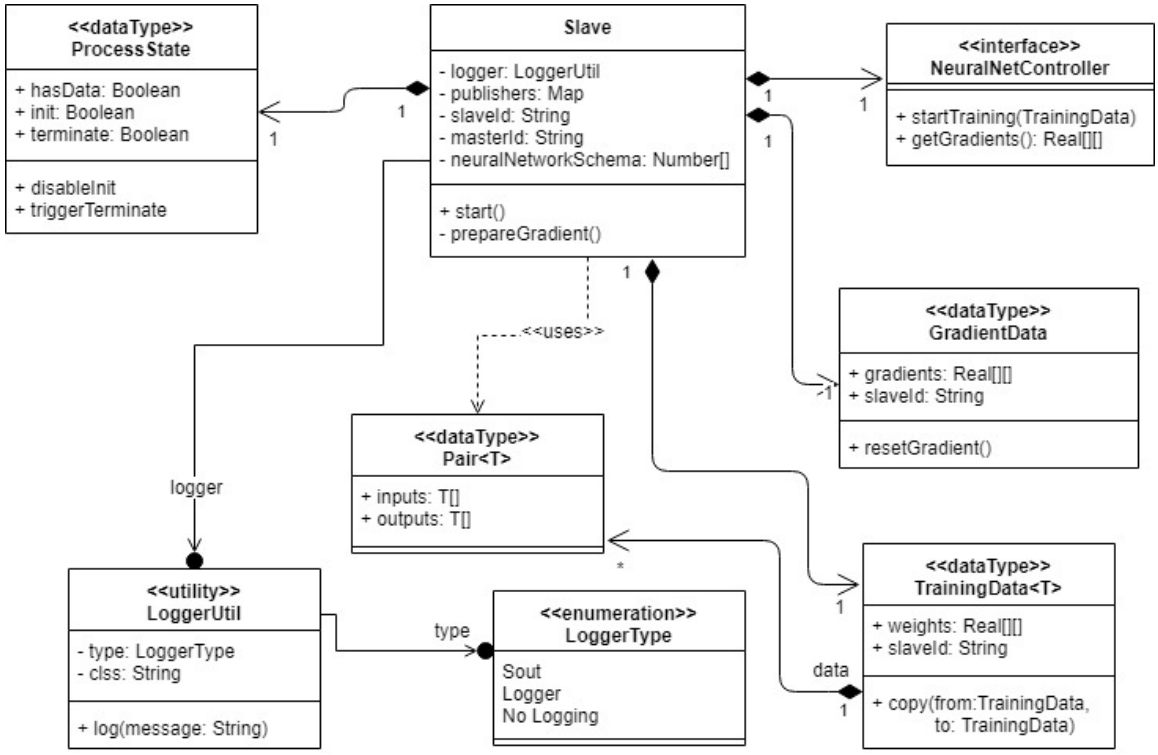
Tato třída slouží jako řídící třída celé aplikace a obsahuje v sobě metodu pojmenovanou *start*, stejně jako v řídícím uzlu, sloužící ke spuštění celého výpočetního uzlu.

V případě třídy *Slave* tato metoda provede inicializaci všech potřebných proměnných včetně neuronové sítě, ale první hodnoty vah přijme až z dat od řídícího uzlu. Inicializuje také vnitřní stavové proměnné a zahájí svůj běh tím, že se dotáže řídícího prvku o data potřebná k výpočtu.

5.4.2 Pomocná třída *ProcessState*

Třída *ProcessState* představuje třídu, která v sobě drží stav výpočtu výpočetního uzlu. Tato třída v sobě obsahuje logické proměnné *hasData*, *init* a *terminate*, které slouží jako návěstí.

Návěstí *hasData* výpočetnímu uzlu říká, že má data pro výpočet a může tedy počítat. Pokud data nemá, musí si o ně říct řídícímu uzlu.



Obrázek 13: Třídní diagram výpočetního uzlu

Návěstí *init* výpočetnímu uzlu naznačuje, že je zatím v init stavu a modifikuje tak některé malé detaily chování.

Poslední návěstí *terminate* pochopitelně značí, že řídící uzel poslal výpočetnímu uzlu ukončovací signál a výpočetní uzel má zahájit proces ukončení.

5.4.3 Třídy *TrainingData*, *GradientData* a *LoggerUtil*

Všechny tyto třídy, které výpočetní uzel během svého běhu využívá, musí být pochopitelně stejné jako ty, které využívá řídící uzel. Nemá tedy smysl tyto třídy znovu popisovat. Informace o třídách *TrainingData*, *GradientData* a *LoggerUtil* lze najít v podkapitole 5.3.

5.5 Využití knihovny Aeron

Využití knihovny Aeron v kódu odpovídá návrhu a není nutné se ní více zabývat. Jediný problém, který nastal při přidávání knihovny Aeron, se nacházel v tom, že Aeron má vícero implementací a originální implementace od společnosti Real Logic není úplně jasně odlišená od dalších implementací a může tedy snadno dojít k záměně.

Originální implementace knihovny Aeron patří v konfiguračním nástroji Maven do skupiny *io.aeron*. Ovšem druhá velmi populární implementace patří do skupiny *uk.co.real-logic*.

Protože Aeron ze skupiny *uk.co.real-logic* používá stejné rozhraní jako originální implementace, může snadno dojít z záměně. Ovšem Aeron ze skupiny *uk.co.real-logic* ukončil svůj vývoj v roce 2015 a funkce, které má poskytovat, nefungují.

Aktuálně prototyp knihovny využívá implementaci knihovny Aeron ze skupiny *io.aeron*, která funguje přesně dle návrhu.

6 Testování

Tato kapitola se zabývá testováním vytvořeného prototypu knihovny. Popisuje testování v lokálním prostředí, testovacím prostředí určeném pro ladění chování knihovny v prostředí podobném produkčnímu prostředí a také v produkčním prostředí na uzlech superpočítače Anselm.

6.1 Testování v lokálním prostředí

První fáze testování probíhala během vývoje knihovny v lokálním prostředí. Lokální testovací prostředí představuje laptop značky Dell Latitude 5580 v nestandardní konfiguraci.

Detail konfigurace lokálního prostředí lze najít v tabulce 1.

Testovací zařízení	
Název	Dell Latitude 5580
OS	Window 10 Enterprise v1709
CPU	Intel(R) Core(TM) i7 - 7820HQ CPU @ 2.90GHz
CPU jádra	4
RAM	32 GB
Disk	NVMe TOSHIBA 512GB

Tabulka 1: Konfigurace lokálního prostředí

Testování v lokálním prostředí probíhalo průběžně po celou dobu vývoje.

Celé toto testování lze rozdělit do tří fází - prototypové, funkční a výkonové. Každá z těchto fází má své specifika, má jiné určení a podává jiné výsledky.

6.1.1 Fáze prototypového testování

Fáze prototypového testování se vyznačuje tím, že je určeno pro testování nově přidaných funkcí knihovny. Ověřuje, že dané funkce je možné do knihovny přidat bez problémů, a že jsou kompatibilní se zbytkem funkcí v knihovně.

Během vývoje bylo do prototypu knihovny přidáno značné množství experimentálních prvků, které buď byly implementovány přímo pro knihovnu nebo pocházely od vývojářů třetích stran.

6.1.2 Fáze funkčního testování

Fáze funkčního testování ověřovala samotnou funkčnost prototypu knihovny. Vždy po přidání určitého množství kódu či nových funkcionalit bylo nutné ověřit, zda knihovna stále vykonává požadované funkce a zda i přidaný kód vykonává svůj úděl správně.

Funkční testování ve většině případů probíhalo formou kouřových testů. Klíčovou funkcionalitu nebylo možné rozumně testovat pomocí unit testů.

6.1.3 Fáze výkonového testování

Po dokončení fází prototypového a funkčního testování, tedy ve chvíli, kdy byl prototyp knihovny bez pochyb funkční a spustitelný, nastal čas pro testování výkonu na lokálním prostředí.

První fáze testů proběhla v lokálním prostředí a byla určena k nastavení hodnot rychlostí výpočtu knihovny nad vlastní testovací vícevrstvou neuronovou sítí. Více než měření rychlostí bylo jejím účelem stanovení standardů a prověření, že časy nejsou viditelně vysoké nebo nízké. Zároveň testovala, že při daném výpočtu dojde i k naučení neuronové sítě.

V tabulce 2 lze nalézt proběhlá měření v lokálním prostředí, výsledné časy a výsledky spuštěné testovací množiny.

Všechny testy proběhly se stejnou konfigurací. Učící poměr byl nastaven na hodnotu 0,6, parametr λ na hodnotu 0,3 a schéma neuronové sítě odpovídalo následujícímu předpisu [2,2,1]. Tedy tři vrstvy v pořadí od vstupní po výstupní vrstvu. Vstupní vrstva obsahovala dva neurony, skrytá vrstva dva neurony a výstupní vrstva jeden neuron.

Každý test byl proveden pětkrát a výsledky jsou mediány naměřených hodnot.

Paralelismus na úrovni jednotlivých výpočetních uzlů nebyl v lokálním prostředí použit, protože při počtu neuronů a vrstev použité sítě se paralelní výpočet vůbec nevyplatí, naopak by přidaná synchronizace vláken způsobovala zpomalení výpočtu.

Číslo testu	**Počet epoch**	**Paralelně**	**Počet výp. uzlů**	**Doba běhu (s)**	**Výsledky***
1	1000	Ne	1	11,43	0/4
2	10000	Ne	1	55,44	2/4
3	100000	Ne	1	489,76	4/4
4	1000	Ne	2	8,55	1/4
5	10000	Ne	2	28,52	3/4
6	100000	Ne	2	206,68	4/4

* Výsledky testování neuronové sítě pomocí testovací množiny vstupů v podobě počet úspěchů / celkový počet vstupů

Tabulka 2: Testování v lokálním prostředí

Z výsledků lokálního testování lze vyvodit následující závěry.

Učení testovací neuronové sítě probíhalo bez problému a dosáhlo očekávaných výsledků. V testech s menším počtem epoch se sice síť nedokázala úspěšné naučit, ale časové výsledky jsou příznivé a příliš se neliší od běhu bez paralelní distribuce.

Při běhu s počtem epoch tisíc a deset tisíc se síť nedokázala úplně správně naučit. Tento výsledek odpovídá testování na neuronové sítí bez paralelní distribuce.

Při běhu s počtem epoch sto tisíc se síť naučit dokázala jak s paralelní distribucí, tak i bez ní.

Pokud se zdvojnásobil počet uzlů, tak se rychlost výpočtu průměrně dvakrát zvýšila. Z trendu rychlosti výpočtu lze vidět, že čím větší je počet epoch, tím větší je rozdíl v rychlosti výpočtu mezi zapojením jednoho a dvou výpočetních uzlů.

6.2 Testování v serverovém prostředí

Testování v serverovém prostředí je část testování, která následovala po úspěšném dokončení testování v lokálním prostředí. Celé testování probíhalo na dvou virtuálních serverech vytvořených univerzitou speciálně pro tento účel. Oba servery mají stejnou konfiguraci, kterou lze vidět v tabulce 3.

Testovací zařízení	
Název	Virtuální server
OS	Ubuntu 18.04.1 LTS
CPU	GenuineIntel QEMU Virtual CPU
CPU jádra	1
RAM	3 GB
Disk	Virtual I/O device 20GB

Tabulka 3: Konfigurace serverového prostředí

Z konfigurace těchto dvou virtuálních serverů lze vidět, že se nejedná o velmi výkonné stroje. Hlavním účelem této části testování bylo ověřit, že komunikace mezi uzly probíhá v pořádku, ale pro porovnání výsledků s lokálními testy byl proveden i výkonnostní test.

6.2.1 Testování komunikace mezi uzly

V této části testování bylo potřeba ověřit, že komunikace mezi řídícím uzel a výpočetními uzly bude fungovat stabilně a spolehlivě do té míry, kterou nám protokol UDP dovoluje. Je třeba ověřit všechny možné varianty a otestovat, že komunikace bude probíhat stále dle předpokladů.

Je zřejmé, že podobný test proběhl už v testování v lokálním prostředí. V lokálním prostředí ovšem neprobíhala komunikace po síti mezi dvěma nezávislými účastníky, ale probíhala mezi dvěma běžícími uzly na jednom testovacím zařízení.

6.2.2 Testování krajních případů

Jako další přišlo na řadu testování krajních případů. Těmi jsou čtyři možné případy. Ve všech případech běží řídící uzel na jednom virtuálním serveru a výpočetní uzel na serveru druhém.

1. Řídící uzel i výpočetní uzel dokončí celý výpočet bez potíží.
2. Řídící uzel z komunikace vypadne, výpočetní uzel běží bez potíží.
3. Řídící uzel běží bez potíží, výpočetní uzel z komunikace vypadne.
4. Z komunikace vypadnou oba uzly.

V prvním případě proběhl test tak, že se zvládl jeden celý výpočet dokončit. Celý výpočet proběhl úspěšně, během komunikace nenastala žádná chyba. Cely proces se opakoval pětkrát.

Ve druhém případě výpadek řídící uzlu znamená selhání výpočtu. Bohužel prototyp knihovny neobsahuje funkční přístup k automatickému restartu řídícího uzlu, a proto pokud se uzel nenastartuje ručně, je výpočet ztracen. Ruční start řídícího uzlu vede ke ztrátě postupu učení. S tímto omezením bohužel musí centralizovaný výpočet počítat.

Ve třetím případě výpadek výpočetního uzlu neznamená konec výpočtu, pouze jeho zdržení. Vzhledem k distribuci naučených hodnot z řídícího uzlu směrem k výpočetnímu uzlu stačí výpočetní uzel znovu nahodit a výpočet může pokračovat bez problémů. Systém automatického startování výpočetních uzlů v knihovně není přítomen.

V případě, že je uzlů více, pokračuje výpočet dále a není vyloženě nutné uzel znovu nastartovat. Z důvodu efektivity výpočtu je to ovšem doporučeno.

Ve čtvrtém a posledním případě pád jak řídícího tak výpočetního uzlu znamená iminentní neúspěšný konec výpočtu. Podobně jako pád řídícího uzlu se i tento stav již nedá zvrátit.

Ve všech případech se prototyp knihovny choval dle očekávání a testování bylo úspěšné.

6.2.3 Testování zpomalení uzlů

V této části testování v serverovém prostředí bylo ověřeno, jestli se prototyp knihovny dokáže vypořádat se zpomalením jednoho z uzlů. Zpomalení výpočtu může mít značný dopad v případě jak řídícího tak výpočetního uzlu.

Testování bylo rozděleno do tří částí. Ve všech případech bylo zpomalení simulováno vložením pauz do zdrojového kódu daného uzel. Délka trvání pauz byla nastavena náhodně.

1. Zpomalení výpočetního uzlu.
2. Zpomalení řídícího uzlu.
3. Zpomalení obou uzlů.

V prvním případě nemá zpomalení příliš velké následky. Vzhledem k tomu, že výpočet výpočetního uzlu může trvat různě dlouho v závislosti na použité neuronové síti, jakékoliv zpomalení nemá na výpočet vliv.

Když výpočetní uzel počítá dlouho, řídící uzel si tohoto uzlu aktivně nevšímá a čeká na informaci o tom, že svůj výpočet ukončil.

V druhém případě má zpomalení větší následky. Pokud by zpomalený řídící uzel minul informaci o dokončení výpočtu od výpočetního uzlu a nevěděl o tom, že mu má poslat nová data, tak by se výpočetní uzel mohl zaseknout v cyklu čekání na data od řídícího uzlu. Proto každý výpočetní uzel obsahuje mechanismus, který zaručuje, že pokud po dotázání se řídícího uzlu nedojdou v daném čase data, zeptá se uzel znovu.

Posledním případ je kombinací obou předchozích případů. Vzhledem k jejich povaze se k nim přistupuje stejně, jako by to byly oddělené případy a jejich řešení je stejné.

Ve všech případech si prototyp knihovny s problémy poradil a testování neodhalilo žádnou chybu.

6.2.4 Výkonnostní testování

Vzhledem k nedostatečnému výkonu virtuálních serverů byly provedeny podobné testy jako v lokálním prostředí. Tyto hodnoty slouží ke zjištění, zda se na jiné sestavě a s komunikací po síti nechová knihovna výrazně odlišně.

Pro testování bylo použito stejné nastavení jako při testování v lokálním prostředí. Tedy učící poměr byl nastaven na hodnotu 0,6, parametr λ na hodnotu 0,3 a schéma neuronové sítě odpovídalo následujícímu předpisu [2,2,1].

V případě jednoho i dvou výpočetních uzlů běžely uzly na jiném serveru než řídící uzel.

Na serverech s takto slabými CPU nedává paralelní testování smysl a nebylo tedy provedeno.

Výsledky testů lze nalézt v tabulce 4.

Číslo testu	**Počet epoch**	**Paralelně**	**Počet výp. uzlů**	**Doba běhu (s)**	**Výsledky***
1	1000	Ne	1	13,46	0/4
2	10000	Ne	1	63,70	1/4
3	100000	Ne	1	498,24	4/4
4	1000	Ne	2	8,82	1/4
5	10000	Ne	2	30,04	3/4
6	100000	Ne	2	221,65	4/4

* Výsledky testování neuronové sítě pomocí testovací množiny vstupů v podobě počet úspěchů / celkový počet vstupů

Tabulka 4: Testování v serverovém prostředí

V porovnání získaných výsledků s výsledky z výkonnostního testování v lokálním prostředí je vidět, že výsledné délky běhu jsou srovnatelné a úměrně pomalejší. Toto zpoždění je způsobeno pomalejší komunikací.

Výsledky testování pomocí testovací množiny dopadlo téměř stejně jako v lokálním prostředí a tedy je vidět, že komunikace přes síť nemá na výsledek vliv.

I testování v serverovém prostředí naznačuje, že rozdíl mezi využitím jednoho a dvou výpočetních jader je přibližně padesát procent. Tento rozdíl se sice natahuje a zkracuje dle počtu epoch, ale neodchyluje se od něj významně.

6.3 Testování na superpočítači

Poslední částí testování bylo spuštění prototypu knihovny na superpočítači za účelem ověření rychlosti a funkčnosti v prostředí, ve kterém by prototyp knihovny měl dosáhnout nejlepších výsledků. Zároveň nám prostředí superpočítače dovoluje volně konfigurovat prostředí, ve kterém bude knihovna provádět výpočet.

Pro testování byl zřízen přístup na oba superpočítače Anselm i Salomon. Vybrán byl superpočítač Anselm. Z pohledu potřebného testování mají oba superpočítače dostatečné parametry a Anselm byl vybrán náhodně.

6.3.1 Superpočítač Anselm

Anselm je superpočítačový klastr uvedený do chodu v roce 2013 provozovaný národním superpočítačovým centrem IT4Innovations.

Klastr Anselm je tvořen 209 výpočetními uzly. Dohromady obsahují 3344 výpočetních jader s 15 TB RAM. Poskytuje dohromady teoretický výkonností limit 94 TFLOP/s. Každý uzel je výkonný 64 bitový počítač, vybavený 16 jádry s alespoň 64 GB RAM. Uzly jsou propojeny pomocí plně neblokující InifiBand sítě a jsou vybaveny Intel Sandy Bridge procesory. Několik uzlů také disponuje NVIDIA Kepler GPU nebo Intel Xeon Phi MIC akcelerátory.[9]

Konfigurace použitých uzlů superpočítače Anselm lze najít v tabulce 5.

Testovací zařízení	
Název	Uzel superpočítače Anselm
OS	CentOS 7
CPU	2x Intel Sandy Bridge E5-2665
CPU jádra	8
RAM	64 GB
Disk	500GB SATA 2,5” 7,2 krpm HDD

Tabulka 5: Konfigurace produkčního prostředí

Konfigurace uzlů superpočítače jasně ukazuje, že se oproti testování v serverovém prostředí jedná o velmi výkonné stroje. Během výpočtu se využívá různé množství uzlů, které jsou přiřazovány při vytvoření práce superpočítačem.

Hlavním účelem této testovací části je ověřit, jak se síť chová na velmi výkonných strojích.

6.3.2 Testování konfigurace s malou neuronovou sítí

Pro porovnání výsledků s předchozím testováním se provedlo testování se stejnou konfigurací jako byla ve výkonnostních testech ostatních prostředí.

Byla použita stejná konfigurace jako při testování v lokálním a serverovém prostředí. Tedy učící poměr byl nastaven na hodnotu 0,6, parametr λ na hodnotu 0,3 a schéma neuronové sítě odpovídalo následujícímu předpisu [2,2,1].

Z důvodu možnosti porovnávat výsledky nebylo využito paralelizace na jednotlivých výpočetních uzlech.

Výsledky testů si lze prohlédnout v tabulce 6.

Tyto hodnoty jsou v porovnání s testy v serverovém i lokálním prostředí očekávatelné. Spuštěním stejných testů jsme v prostředí superpočítače získali lepší výsledky.

V testech s tisícem epoch se hodnoty liší o málo. Rozdíly jsou v jednotkách sekund, nehledě na množství výpočetních uzlů. Mezi distribucí s čtyřmi a distribucí s šesti výpočetními uzly je rozdíl pouhých 0.01 sekundy. Tedy zvětšování distribuce pro takto malou síť s takovým počtem epoch nemá význam.

Číslo testu	Počet epoch	Paralelně	Počet výp. uzlů	Doba běhu (s)	Výsledky*
1	1000	Ne	1	7,16	0/4
2	10000	Ne	1	10,17	1/4
3	100000	Ne	1	33,20	4/4
4	1000	Ne	2	6,17	1/4
5	10000	Ne	2	8,15	3/4
6	100000	Ne	2	19,17	4/4
7	1000	Ne	4	6,18	1/4
8	10000	Ne	4	7,17	3/4
9	100000	Ne	4	13,17	4/4
10	1000	Ne	6	6,15	1/4
11	10000	Ne	6	7,16	3/4
12	100000	Ne	6	11,18	4/4

* Výsledky testování neuronové sítě pomocí testovací množiny vstupů v podobě počet úspěchů / celkový počet vstupů

Tabulka 6: Testování malé neuronové sítě v produkčním prostředí

V testech s deseti tisíci epoch se již výsledné časy liší výrazněji. Prototyp knihovny dosahuje na superpočítači výrazně lepších časů a zlepšují se i při využití více výpočetních uzlů. Rozdíl není stabilní, ale vždy při přidání dvou dalších výpočetních uzlů je výsledný čas lepší. I když rozdíly jsou v případě velkého množství uzlů zanedbatelné.

Testy se sto tisíci epochami už na superpočítači představují zásadní rozdíl. Testování oproti lokálnímu i serverovému prostředí vykazuje posunutí doby výpočtu z řádů minut do řádů sekund. Efektivita výpočtu pak v tomto případě nabírá zcela jiných rozměrů.

Při celkovém pohledu na testy je vidět, že si prototyp knihovny zachovává své vlastnosti, které byly stanoveny během předchozích testování a na superpočítači se tyto vlastnosti ještě zlepšují.

6.3.3 Testování s velkou neuronovou sítí

Další testování má ukázat chování neuronové sítě při práci se sítí velkých rozměrů. Hlavním účelem této části testování je zjistit, zda je prototyp schopen naučit takto velkou neuronovou síť a v jestli jsou časy učení přijatelné.

Pro testování byla zvolena neuronová sít se dvěmi skrytými vrstvami a v součtu s 1203 neurony. Síť těchto rozměrů nepatří k těm největším sítím využívaným v komerční či akademické sféře, ale je dost velká na to, aby její výpočet nebylo možné dokončit v intervalu pár sekund jako testované sítě v minulých podkapitolách, a zároveň je dost velká na to, aby se u ní dalo využít paralelního zpracování na úrovni jednotlivých výpočetních uzlů.

Přesná konfigurace použité neuronové sítě představuje učící poměr nastavený na hodnotu 0,6, parametr λ na hodnotu 0,2 a schéma neuronové sítě odpovídalo předpisu [2,1000,200,1].

Velikost sítě byla vybrána tak, aby její testovací výpočet nezabral více než hodinu. To znamená, že veškeré testy by trvaly maximálně třicet hodin, a tak by bylo možné provést všechny testy v reálném čase i s rezervou pro neúspěšné pokusy.

Výsledky testování lze vidět v tabulce 7.

Číslo testu	**Počet epoch**	**Paralelně**	**Počet výp. uzlů**	**Doba běhu (min)**	**Výsledky***
1	500	Ne	1	61,58	4/4
2	500	Ne	2	30,75	4/4
3	500	Ne	4	20,18	4/4
4	500	Ne	6	19,74	4/4
5	500	Ano	2	29,45	4/4
6	500	Ano	4	19,46	4/4
7	500	Ano	6	19,51	4/4

* Výsledky testování neuronové sítě pomocí testovací množiny vstupů v podobě počet úspěchů / celkový počet vstupů

Tabulka 7: Testování velké neuronové sítě v produkčním prostředí

Nejdůležitějším poznatkem z této části testování je fakt, že všechny pokusy skončily úspěšným naučením sítě, a tedy prototyp knihovny práci s neuronovými sítě velkých rozměrů.

Dalším důležitým faktem proběhlého testování je ukončení všech testů v reálném čase, který nenaznačuje jakékoliv problémy při běhu.

Při testování případů s a bez využití paralelizace na úrovni jednotlivých výpočetních uzlů bylo zjištěno, že zvolená síť není ještě dostatečně velká, aby rozdíly v čase byly výrazné, ale i na této sítí je vidět, že je možné dosáhnout ještě lepších časů pomocí paralelizace. Je také vidět, že zvolená metoda paralelizace nijak neovlivňuje naučení sítě.

Tato část testování jednoznačně prokázala, že prototyp knihovny je schopen efektivně pracovat i s neuronovými sítěmi s velkým počtem neuronům a zároveň využívat velký počet výpočetních uzlů. I v takovém případě dojde knihovna ke správným výsledkům.

6.3.4 Zrychlení a efektivita paralelizace

Pro ideální znázornění efektivity distribuce výpočtů mezi výpočetní uzly využijeme vlastností zrychlení S_N a efektivity paralelizace ϵ_N paralelních algoritmů. Vlastnost zrychlení S_N lze spočítat následovně:

$$S_N = \frac{\tau_1}{\tau_N} \tag{14}$$

Kde N představuje počet výpočetních uzlů, τ_N představuje délku výpočtu s využitím jednoho výpočetního uzlu a τ_N značí délku výpočtu s využitím N výpočetních uzlů.

Efektivitu paralelizace ϵ_N lze získat dosazením do následující rovnice:

$$\epsilon_N = \frac{S_N}{N} \tag{15}$$

Zde N zastupuje počet výpočetních uzlů a S_N představuje zrychlení.[10]

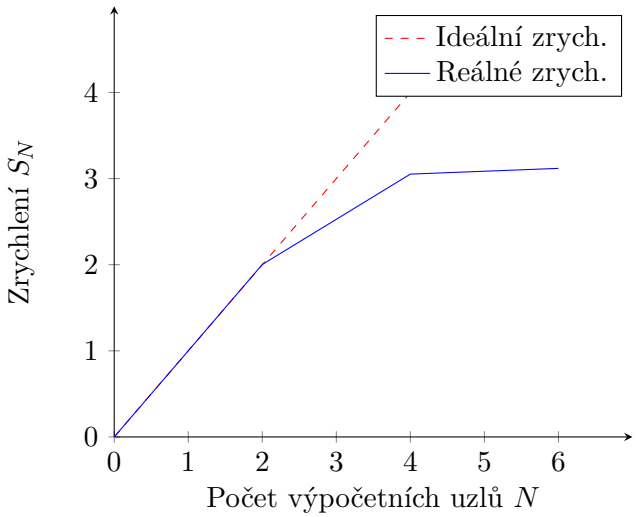
Výsledky získané při testování v produkčním prostředí, viz tabulka 7, představují délky výpočtů τ_N.

Dosazením do rovnice pro výpočet zrychlení S_N byly získány konkrétní hodnoty zrychlení. Hodnoty lze najít v tabulce 8.

Počet výp. uzlů N	**Zrychlení** S_N
1	1.000
2	2.003
4	3.052
6	3.119

Tabulka 8: Hodnoty zrychlení S_N

Závislost hodnoty zrychlení S_N na počtu výpočetních uzlů N si lze prohlédnout na obrázku 14. Pro porovnání je k reálnému zrychlení přidáno ideální zrychlení.



Obrázek 14: Závislost zrychlení S_N na počtu výpočetních uzlů N

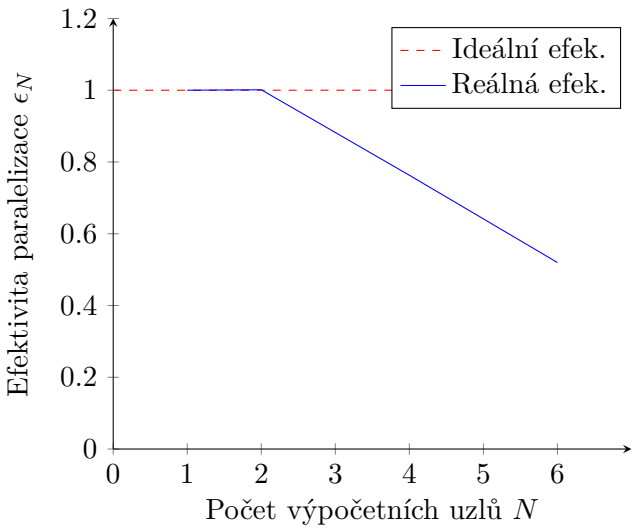
Ze získaných hodnot i ze zpracování závislosti zrychlení na počtu výpočetních uzlů lze vyvodit, že nárůst zrychlení je výrazný až do využití čtyř výpočetních uzlů. Následně přidávání uzlů přináší stále větší režii a tendence růstu zrychlení se začíná snižovat.

Dosazením hodnot z testování do rovnice efektivity paralelizace ϵ_N získáme hodnoty, které lze vidět v tabulce 9.

Závislost efektivity paralelizace ϵ_N na počtu výpočetních uzlů N je možné vidět na obrázku 15. Pro porovnání je k reálné efektivitě paralelizace přidána ideální efektivita paralelizace.

Počet výp. uzlů N	**Efektivita paralelizace ϵ_N**
1	1.000
2	1.001
4	0.763
6	0.520

Tabulka 9: Hodnoty efektivity paralelizace ϵ_N



Obrázek 15: Závislost efektivity ϵ_N na počtu výpočetních uzlů N

Výsledné hodnoty efektivity paralelizace představují potvrzení výsledků výpočtu zrychlení. Efektivita paralelizace si udržuje přijatelně vysokou hodnotu do využití čtyř výpočetních uzlů včetně. Následně začíná efektivita klesat do nižších hodnot kvůli zvyšující se režii komunikace mezi větším množstvím výpočetních uzlů.

7 Závěr

Cílem této diplomové práce bylo prozkoumat aktuální možnosti distribuce výpočtů nad vícevrstvými neuronovými sítěmi a navrhnout a implementovat prototyp knihovny pro aplikaci Modeler neuronových sítí, který jí umožní distribuované výpočty využívat.

Analýza dostupných možností distribuce výpočtů ukázala, že toto téma je v odborné sféře široce probíráno a existuje vícero způsobů jak k distribuci výpočtu přistupovat. Všechny možnosti byly podrobeny zkoumání a bylo vybráno řešení poskytující ideální parametry a splňující všechny požadavky. Součástí zkoumání dostupných možností byla i analýza aktuálního stavu knihoven s otevřeným kódem řešících distribuci výpočtu. Ukázalo se, že je dostupné pouze malé množství takových knihoven.

Pomocí informací získaných během analýzy byla navržena architektura a funkcionalita prototypu knihovny tak, aby dokázala efektivně distribuovat výpočet nad vícevrstvými neuronovými sítěmi ve všech požadovaných prostředích. Návrh knihovny musel řešit všechny problémy spojené s distribucí výsledků výpočtů, komunikací mezi zúčastněnými uzly a řízením celého výpočtu.

Následně byl prototyp knihovny implementován přesně dle návrhu a byly vyřešeny všechny implementační problémy, které během implementace vznikly a s nimiž návrh nepočítal.

Poté, co byl prototyp knihovny úspěšně vytvořen, proběhlo jeho testování. Knihovna byla testována v různých prostředí a s použitím různých konfigurací tak, aby se ověřilo, že splňuje veškeré požadavky a dokáže správně distribuovat výpočty. Součástí testování bylo také spuštění série testů na superpočítači tak, aby se ověřilo chování knihovny i na stroji umožňujícím distribuci na velký počet uzlů.

Prototyp knihovny tedy splnil zadané požadavky a je připraven k použití.

Literatura

[1] GUPTA, Suyog, Wei ZHANG a Fei WANG. *Model Accuracy and Runtime Tradeoff in Distributed Deep Learning:A Systematic Study*[online], IBM T. J. Watson Research Center, Yorktown Heights, NY, 2015, [cit. 27. března 2019], dostupné z: `https://arxiv.org/abs/1509.04210`

[2] LUDVIG, Ericson a Mbuvha RENDANI. *On the Performance of Network Parallel Training in Artificial Neural Networks*[online], KTH Royal Institute of Technology, Stockholm, Sweden, 2017, [cit. 14. dubna 2019], dostupné z: `https://arxiv.org/abs/1701.05130`

[3] BEN-NUN, Tal, Torsten HOEFLER. *Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis*[online], ETH, Zurich, Switzerland, 2018, [cit. 14. dubna 2019], dostupné z: `https://arxiv.org/abs/1802.09941`

[4] DL4J Distributed Training: Technical Explanation [online], [cit. 28. března 2019], dostupné z: `https://deeplearning4j.org/docs/latest/deeplearning4j-scaleout-technicalref`

[5] Deep Learning for Java [online], [cit. 30. března 2019], dostupné z: `https://deeplearning4j.org/`

[6] Aeron knihovna, [cit. 8. dubna 2019], dostupné z: `https://github.com/real-logic/aeron`

[7] ROJAS, Raúl, *Neural networks: a systematic introduction*, New York: Springer-Verlag, 1996, ISBN 3-540-60505-3

[8] GOODFELLOW, Ian, Yoshua BENGIO a Aaron COURVILLE. *Deep learning*, Cambridge, Massachusetts: The MIT Press, 2016, ISBN 9780262035613

[9] IT4Innovations Documentation [online], [cit. 19. dubna 2019], dostupné z: `https://docs.it4i.cz/anselm/introduction/`

[10] Efficiency of Parallel Algorithm [online], [cit. 23. dubna 2019], dostupné z: `https://www.sdsc.edu/ majumdar/thesis/node47.html`

A Obsah elektronické přílohy

```
aeron
├── master
│   ├── src/main/java/cz/pus0049/aeron/master
│   │   ├── config
│   │   ├── GradientData.java
│   │   ├── LoggerUtil.java
│   │   ├── Master.java
│   │   ├── Pair.java
│   │   ├── SendState.java
│   │   └── TrainingData.java
│   ├── src/main/resources
│   │   └── config-master.yml
│   └── pom.xml
├── slave
│   ├── src/main/java/cz/pus0049/aeron/slave
│   │   ├── config
│   │   ├── neuralnet
│   │   ├── GradientData.java
│   │   ├── LoggerUtil.java
│   │   ├── NeuralNetController.java
│   │   ├── Pair.java
│   │   ├── ProcessState.java
│   │   ├── Slave.java
│   │   └── TrainingData.java
│   ├── src/main/resources
│   │   └── config-slave.yml
│   └── pom.xml
├── pom.xml
└── README.md
```

B Návod ke spuštění knihovny

Totožný návod lze nalézt v README.md souboru v kořenovém adresáři elektronické přílohy.

Prerekvizity:

1. Java SDK ve verzi 11.0.2 (LTS)
2. Apache Maven 3.6.0

Postup spuštění:

1. Dodejte vlastní implementaci neuronové sítě, lze využít i přiloženou testovací síť.
2. Nastavte konfigurační parametry v souborech *master/. . . /resources/config-master.yml* a *master/. . . /resources/config-slave.yml*.
3. Nastavte konfigurační parametry maven profilů v *master/pom.xml* a *slave/pom.xml*.
4. Spusťte v terminálu příkaz *mvn clean install -P<dev|test|prod>*.
5. Spustitelné soubory JAR se vytvoří ve složce *master/target* a *slave/target*.
6. Spusťte aplikace následovně (použití Java argumentů přepíše části konfigurace z bodů 2. a 3.):
   (a) bez použití Java argumentů: *java -jar master-<aktuální-verze>.jar* a *java -jar slave-<aktuální-verze>.jar*.
   (b) s použitím Java argumentů: *java -jar master-<aktuální-verze>.jar <master-ip:port> <slave1-ip:port,slave2-ip:port,. . . > <počet-epoch>* a *java -jar slave-<aktuální-verze>.jar <master-ip:port> <slave-ip:port> <paralelizace-true|false>*.